SYLLABI-BOOK MAPPING TABLE
JAVAPROGRAMMING

JAVA PROGRAMMING

Course Material Prepared by
Dr.S. MEGANATHAN Ph.D.
Senior Assistant Professor
Department of Computer Science
SASTRA University
Kumbakonam-612001.

UNIT I: JAVA EVOLUTION

1.1 INTRODUCTION

Java is an object-oriented programming language with its runtime environment. It is a combination of features of C and C++ with some essential additional concepts. Java is well suited for both standalone and web application development and is designed to provide solutions to most of the problems faced by users of the internet era. Java was developed by Sun Microsystems Inc in 1991, later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton. It is a simple programming language. Writing, compiling and debugging a program is easy in java. It helps to create modular programs and reusable code.

1.2 OBJECTIVES

Understand fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries, etc.

1.3 JAVA HISTORY

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of java starts with Green Team. Java team members (also known as **Green Team**), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called "Greentalk" by James Gosling, and file extension was .gt.

4) After that, it was called Oak and was developed as a part of the Green project.

5) Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

1.4 JAVA FEATURES

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java buzzwords.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

I. Runtime Environment
II. API(Application Programming Interface)

1. **Secured** - Java is best known for its security. With Java, we can develop virus-free systems

    - No explicit pointer
    - Java Programs run inside a virtual machine sandbox
    - Classloader
    - Bytecode Verifier
    - Security Manager

2. **Robust** - Robust simply means strong. There is a lack of pointers that avoids security problems, There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
3. **Architecture neutral** - Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
4. **Compiled and Interpreted –** Java Compiler translates Source code into what is known as bytecode instruction. Byte code are not machine instructions, java interpreter generates machine code that can be directly executed by the machine that is running the java program, java is a both compiled and interpreted language.
5. **High Performance** - Java is faster than other traditional interpreted programming languages because Java byte code is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).
6. **Multithreaded -** A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread.

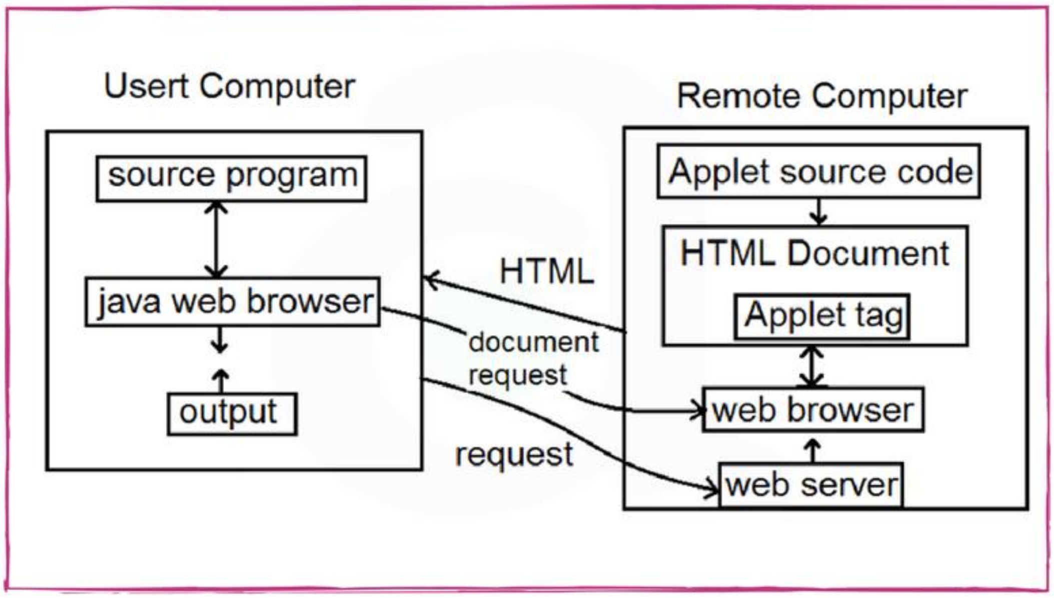
7. **Distributed -** Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
8. **Dynamic -** Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

1.5 JAVA AND INTERNET

- Java is strongly associated with the internet because of the first application program is written in Java was hot Java.
- Web browsers to run applets on the internet.
- Internet users can use Java to create applet programs & run then locally using a Java-enabled browser such as hot Java.
- Java applets have made the internet a true extension of the storage system of the local computer.

1.6 WWW

- World wide web is a collection of information stored on internet computers.
- World wide web is an information retrieval system designed to be used in the internet's distributed environment.
- World wide web contains web pages that provide both information and controls.
- Web pages contain HTML tags that enable us to find retrieve, manipulate and display documents world wide.
- Before Java, the world wide web was limited to the display of still images & texts.
- With the help of Java WWW is capable of supporting animation graphics, games and wide rage special effects.



Java and World Wide Web (www)

1.7 WEB BROWSER

The internet is a vast sea of information represented in many formats and stored on many computers. a browser is a software application used to locate, retrieve and display content on the World Wide Web, including Web pages, images, video and other files.

An example of Web Browsers: Hot Java, Netscape Navigator, Internet Explorer, Google Chrome

Web Browser - local computer should be connected to the internet

Web Server - A program that accepts a request from a user and gives output as per the requirement. Apache TomCat server is one of the major web servers.

Web Browser - The web browser is a software that will allow you to view web pages on the internet. it is a program that you use to access the Internet. A program that provides the access of WWW and runs java applets. Chrome and Firefox are two major web browsers.

HTML - HTML is short for HyperText Markup Language. HTML is used to create electronic documents (called pages) that are displayed on the World Wide Web. Each page contains a series of connections to other pages called hyperlinks.

APPLET tag - The HTML tag specifies an applet. It is used for embedding a java applet within an HTML Document. It is not Supported in HTML

Java Code - Java code is used for defining Java APPLETS

Byte Code - Compiled java code that is referred to in the applet tag and transfers to the user computer.

Proxy Server - An intermediate server between the requesting client work station and the original server. It is typically implemented for ensuring security.

Mail Server - A mail server (also known as a mail transfer agent or MTA, a mail transport agent, a mail router or an Internet mailer) is an application that receives an incoming e-mail from local users (people within the same domain) and remote senders and forwards outgoing e-mail for delivery.

UNIT II: OVERVIEW

2.1SIMPLE JAVA PROGRAM

```
class Simple{
public static void main(String args[]){
System.out.println("Hello Java");
}
}
```

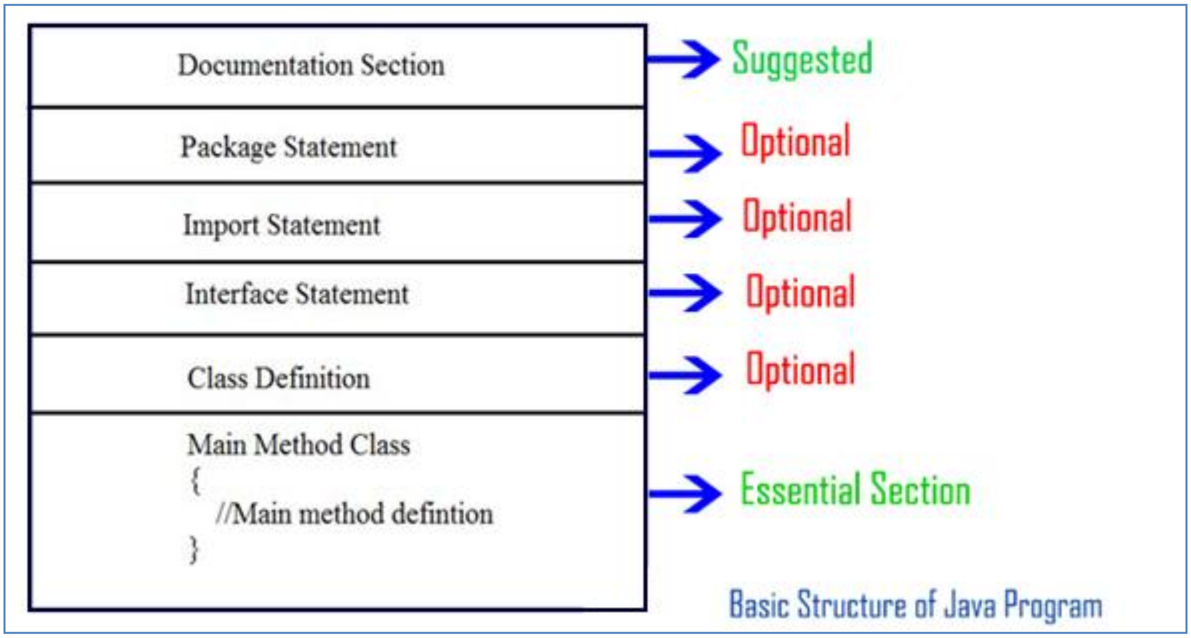
To Compile: javac simple.java

To execute: java simple

Output: Hello Java

2.2PROGRAM STRUCTURE

A Java program consists of different sections. Some of them are mandatory but some are optional. The optional section can be excluded from the program depending upon the requirements of the programmer.



Documentation Section:

It includes the comments that improve the readability of the program. A comment is a non-executable statement that helps to read and understand a program especially when your programs get more complex. It is simply a message that exists only for the programmer and is ignored by the compiler.

Single Line

It Starts with a Double Slash Symbol (//).

//Calculate sum of two numbers

Multiline Comment

Java Programmer can use C/C++ Comment Style that begins with delimiter /* and ends with */.

/*calculate sum of two numbers

and it is a multiline comment */

Documentation Comments:

This Comment style is new in java. Such comments begin with /** and end with */

/** The text enclosed here will be part of program documentation */

Package Statement:

A package is a collection of classes, interfaces and sub-packages. A sub package contains collection of classes, interfaces and sub-sub packages etc. java.lang.*; package is imported by default and this package is known as default package. It must appear as the first statement in the source code file before any class or interface declaration. This statement is optional.

```
package institute;
```

Import statements:

Java contains many predefined classes that are stored into packages. In order to refer these standard predefined classes in your program, you need to use fully qualified name (i.e. Packagename.className).

```
import java.util.Date;
```

Interface Section

In the interface section, we specify the interfaces. An interface is similar to a class but contains only constants and method declarations. Interfaces cannot be instantiated. They can only be implemented by classes or extended by other interfaces.

```
interface stack
{
void push(int item); // Insert item into stack

int pop(); // Delete an item from stack
}
```

Class Definition:

Java program may contain multiple class definition. Classes are primary feature of Java program. The classes are used to map real world problems.

```
class Addition
{ void add(String args[])
   {    int a=2, b=3, c;
    c=a+b;    System.out.println(c);
    } }
```

Main Method Class Section:

The Class section describes the information about user-defined classes present in the program. A class is a collection of fields (data variables) and methods that operate on the fields. Every program in Java consists of at least one class, the one that contains the main method.

2.3TOKENS

When we write a program, we need different important things. We require language tokens, white spaces, and formats.

There are various tokens used in Java:

- Keywords
- Identifiers
- Literals
- Operators
- Separators

Keywords:

Keywords are an essential part of a language definition. These Keywords, combined with Operators and Separators, Java language has reserved 50 words as keywords

- Byte Abstract
- Class
- Do
- Extends

Identifiers:

Identifiers are programmer designed tokens. They are used for naming Classes, methods, Variables, Objects, labels, Packages and interfaces in program.

- They can have alphabets, digits, and the underscore and doller sign characters.
- They must not begin digit.
- Uppercase and lowercase letters are distinct.
- They can be of any length.

Literals:

Literals in java are a sequence of characters(digits, letters and other characters) that represent constants values to be stored in variables. They are five major types:

- Integer Literals
- Floating Point Literals
- Character Literals
- String Literals
- Boolean Literals

Operators:

An Operator is a symbol that takes one or more arguments and Operators on them to produce a result

Separators:

Separators are symbols used to indicate where groups of code are divided and arranged.

2.4JAVA STATEMENT

A statement specifies an action in a Java program, such as assigning the sum of x and y to z, printing a message to the standard output, writing data to a file, etc.

Statements in Java can be broadly classified into three categories:

- Declaration statement
- Expression statement
- Control flow statement

Declaration Statement:

A declaration statement is used **to declare a variable**. For example,

```
int num;

int num2 = 100;

String str;
```

Expression statement:

An expression with a semicolon at the end is called an expression statement. For example,

/Increment and decrement expressions

```
num++;

++num;

num--;

--num;          //Assignment expressions

num = 100;

num *= 10;  //Method invocation expressions

System.out.println("This is a statement");

someMethod(param1, param2);
```

Control flow statement:

By default, all statements in a Java program are executed in the order they appear in the program. Sometimes you may want to execute a set of statements repeatedly for a number of times or as long as a particular condition is true.

All of these are possible in Java using control flow statements. If block, while loop and for loop statements are examples of control flow statements.

UNIT III:WRITING JAVA PROGRAMS

3.1 JVM

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

Java Virtual Machine and it exists only inside the computer memory.



3.2 CONSTANTS

Constants in java are fixed values those are not changed during the Execution of program. A literal is a constant value that can be classified as integer literals, string literals and boolean literals.

Integer Constants:

- Integer Constants refers to a Sequence of digits. An Integer Constant must have at Least one Digit.
- It must not have a Decimal value.
- It could be either positive or Negative.
- If no sign is Specified then it should be treated as Positive.
- No Spaces and Commas are allowed in Name.

Real Constants:

- A Real Constant must have at Least one Digit.
- It must have a Decimal value.
- It could be either positive or Negative.
- If no sign is Specified then it should be treated as Positive.

- No Spaces and Commas are allowed in Name.

Single Character Constants:

A Character is Single Alphabet a single digit or a Single Symbol that is enclosed within Single inverted commas.

Like 'S' ,'1' etc are Single Character Constant.

String Constants:

String is a Sequence of Characters Enclosed between double Quotes These Characters may be digits, Alphabets Like "Hello" , "1234" etc.

Backslash Character Constants:

Java Also Supports Backslash Constants those are used in output methods For Example \n is used for new line Character These are also Called as escape Sequence or backslash character Constants.

3.3 VARIABLES

Variable is name of reserved area allocated in memory, A variable is a container which holds the value while the java program is executed. A variable is assigned with a datatype.

There are three types of variables in java: local, instance and static.

Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

Instance Variable

A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static. It is called instance variable because its value is instance specific and is not shared among instances.

Static variable

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

```
class A{

int data=50;//instance variable
```

```
static int m=100;//static variable

void method(){

int n=90;//local variable

} }//end of class
```

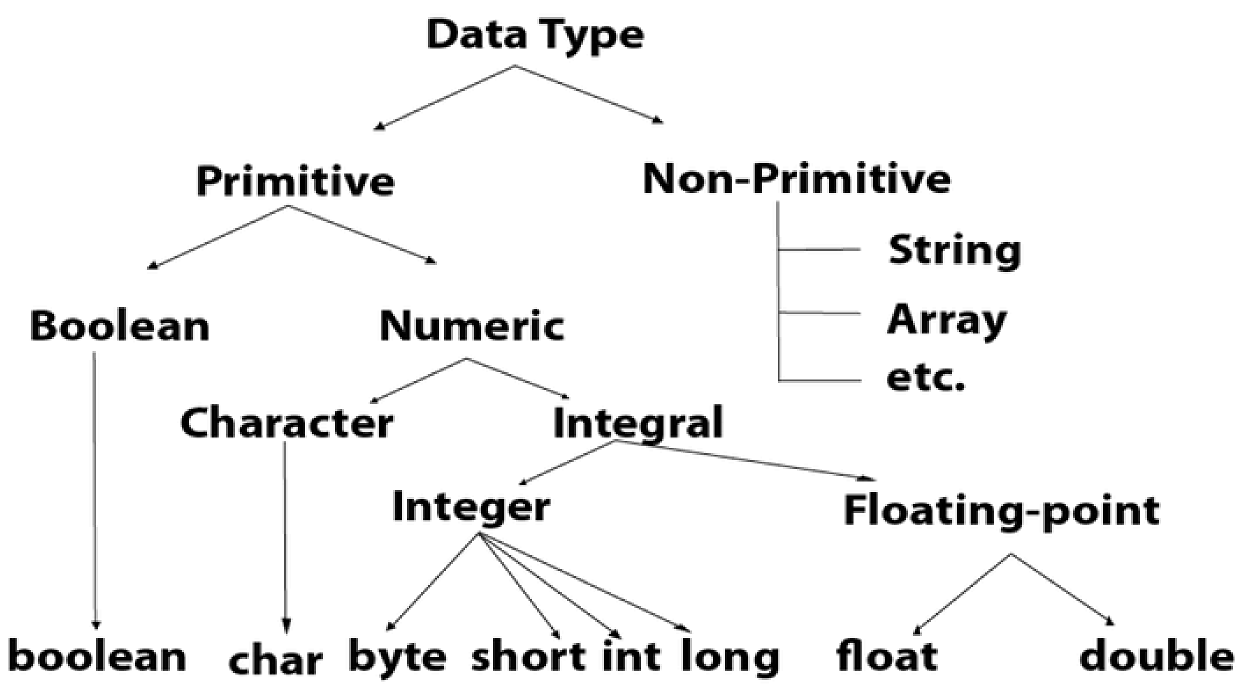
3.4 DATA TYPE

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Primitive Data Types:

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.



Boolean Data Type:

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

Example: Boolean one = false

Byte Data Type:

It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

Example: byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default valvue is 0.

Example: short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

Example: int a = 100000, int b = -200000

Long Data Type

1.1 The long data type is a 64-bit two's complement integer. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers.

Example: float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example: char letter = 'A'

3.6 TYPE CASTING

Type casting is used to convert an object or variable of one type into another.

dataType variableName = (dataType) variableToConvert;

Example:

```
double calculatedMark = 87.6;

int finalGrade = (int)calculatedMark; //will return 87
```

UNIT IV: OPERATORS

4.1 ARITHMETIC

The Java programming language supports various arithmetic operators for all floating-point and integer numbers. These operators are + (addition), - (subtraction), * (multiplication), / (division), and % (modulo).

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

4.2 RELATIONAL

Java assignment operator is one of the most common operator. It is used to assign the value on its right to the operand on its left. These Operators <,>,<=,>=,== and !=.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

4.3 LOGICAL

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition, Logical operators in java are: &&, ||, !

```
public class LogicalOperatorDemo {
public static void main(String args[]) {
boolean b1 = true;
boolean b2 = false;
System.out.println("b1 && b2: " + (b1&&b2));
System.out.println("b1 || b2: " + (b1||b2));
System.out.println("!(b1 && b2): " + !(b1&&b2));
}
}
```

4.4 ASSIGNMENT

Assignments operators in java are: =, +=, -=, *=, /=, %=, Assignments operators are used to assign the value of an expression to a variable.

```
public class AssignmentOperatorDemo {
public static void main(String args[]) {
int num1 = 10;
int num2 = 20;
num2 = num1;
System.out.println("= Output: "+num2);
 num2 += num1;
System.out.println("+= Output: "+num2);
 num2 -= num1;
System.out.println("-= Output: "+num2);
num2 *= num1;
System.out.println("*= Output: "+num2);
 num2 /= num1;
System.out.println("/= Output: "+num2);
num2 %= num1;
System.out.println("%= Output: "+num2);
}
}
```

4.5 INCREMENT AND DECREMENT

Java has two very useful operators not generally found in many other language.

++ and –

```
public class AutoOperatorDemo {
public static void main(String args[]){
int num1=100;
int num2=200;
num1++;
num2--;
System.out.println("num1++ is: "+num1);
System.out.println("num2-- is: "+num2);
}
}
```

4.6 CONDITIONAL

The Java Conditional Operator selects one of two expressions for evaluation, which is based on the value of the first operands.

Syntax

```
expression1 ? expression2:expression3;
```

```
public class condiop {
public static void main(String[] args) {
String out;
int  a = 6, b = 12;
out = a==b ? "Yes":"No";
System.out.println("Ans: "+out);
}
}
```

4.7 BITWISE OPERATORS

There are six bitwise Operators: &, |, ^, ~, <<, >>

```
num1 = 11; /* equal to 00001011*/
num2 = 22; /* equal to 00010110 */
```

```
public class BitwiseOperatorDemo {
public static void main(String args[]) {
int num1 = 11;  /* 11 = 00001011 */
int num2 = 22;  /* 22 = 00010110 */
int result = 0;
result = num1 & num2;
System.out.println("num1 & num2: "+result);
result = num1 | num2;
System.out.println("num1 | num2: "+result);
result = num1 ^ num2;
System.out.println("num1 ^ num2: "+result);
result = ~num1;
System.out.println("~num1: "+result);
result = num1 << 2;
System.out.println("num1 << 2: "+result); result = num1 >> 2;
System.out.println("num1 >> 2: "+result);
}
}
```

4.8 SPECIAL OPERATORS

Java supports some special operators of interest such as instance of operator and member selection operator(.).

Syntax:

```
object-reference instanceof type;;
```

Example:

```
class Company {}
public class Employee extends Company {
public void check() {
System.out.println("Success.");
```

```
        }
         public static void view(Company c) {
         if (c instanceof Employee) {
        Employee b1 = (Employee) c;
         b1.check();
      }
   }
        public static void main(String[] args) {
        Company c = new Employee();
         Employee.view(c);
   }
}
```

UNIT V: EXPRESSIONS

5.1 ARITHMETIC

Arithmetic expressions in Java are composed with the usual operators +, –, *, / and the remainder operator %. Multiplication and division operations have higher priority than addition and subtraction. Operations with equal priority are performed from left to right. Parenthesis are used to control the order of operation execution.

$a + b / c \equiv a + (b / c)$

$a * b - c \equiv (a * b) - c$

$a / b / c \equiv (a / b) / c$

Example

```
class example
{
    public static void main ( String[] args )
{
    long x ; //a declaration without an initial value

    x = 123; //an assignment statement
    System.out.println("The variable x contains: " + x );
}
}
```

5.2 EVALUATION OF EXPRESSION

Expressions are evaluated using an assignment statement of the form, Java applications process data by evaluating expressions, which are combinations of literals, method calls, variable names, and operators. Evaluating an expression typically produces a new value, which can be stored in a variable, used to make a decision, and so on.

Variable=expression;

X=a*b-c;

Y=b/c*a;

Z=a-b/c+d;

5.3 OPERATOR PRECEDENCE AND ASSOCIATIVELY

Java operators have two properties those are precedence, and associativity. Precedence is the priority order of an operator, if there are two or more operators in an expression then the operator of highest priority will be executed first then higher, and then high. For example, in expression 1 + 2 * 5, multiplication (*) operator will be processed first and then addition. It's because multiplication has higher priority or precedence than addition.

Precedence	Operator	Description	Associativity
1	[] () .	array index method call member access	Left -> Right
2	++ -- + - ~ !	pre or postfix increment pre or postfix decrement unary plus, minus bitwise NOT logical NOT	Right -> Left
3	(type cast) new	type cast object creation	Right -> Left
4	* / %	multiplication division modulus (remainder)	Left -> Right
5	+ - +	addition, subtraction string concatenation	Left -> Right
6	<< >> >>>	left shift signed right shift unsigned or zero-fill right shift	Left -> Right

<table>
<tr><td>7</td><td><
<=
>
>=
instanceof</td><td>less than
less than or equal to
greater than
greater than or equal to
reference test</td><td>Left -> Right</td></tr>
<tr><td>8</td><td>==
!=</td><td>equal to
not equal to</td><td>Left -> Right</td></tr>
<tr><td>9</td><td>&</td><td>bitwise AND</td><td>Left -> Right</td></tr>
<tr><td>10</td><td>^</td><td>bitwise XOR</td><td>Left -> Right</td></tr>
<tr><td>11</td><td>|</td><td>bitwise OR</td><td>Left -> Right</td></tr>
<tr><td>12</td><td>&&</td><td>logical AND</td><td>Left -> Right</td></tr>
<tr><td>13</td><td>||</td><td>logical OR</td><td>Left -> Right</td></tr>
<tr><td>14</td><td>? :</td><td>conditional (ternary)</td><td>Right -> Left</td></tr>
<tr><td>15</td><td>=
+=
-=
*=
/=
%=
&=
^=
|=
<<=
>>=
>>>=</td><td>assignment and short hand assignment operators</td><td>Right -> Left</td></tr>
</table>

UNIT VI: DECISION MAKING AND BRANCHING

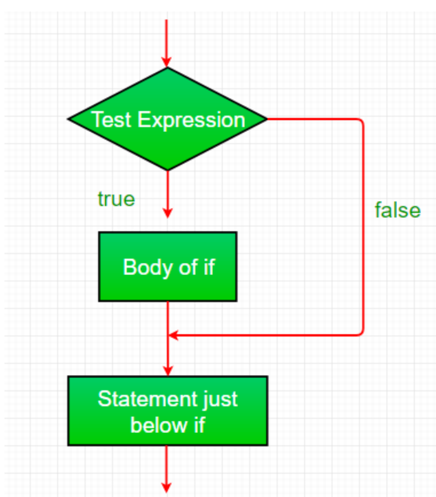
6.1 IF

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Flow Chart



Example:

```
class IfDemo
{
  public static void main(String args[])
  {
   int i = 10;
   if (i > 15)
  System.out.println("10 is less than 15");
// This statement will be executed
// as if considers one statement by default
 System.out.println("I am Not in if");
  }
}
```

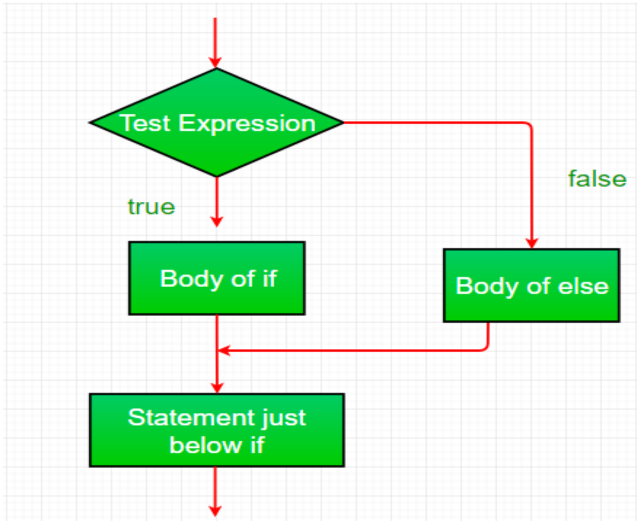
6.2 IF ELSE

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement.

Syntax

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

Flow Chat:



Example:

```
class IfElseDemo
{
public static void main(String args[])
{
int i = 10;
if (i < 15)
System.out.println("i is smaller than 15");
else
System.out.println("i is greater than 15");
}
}
```

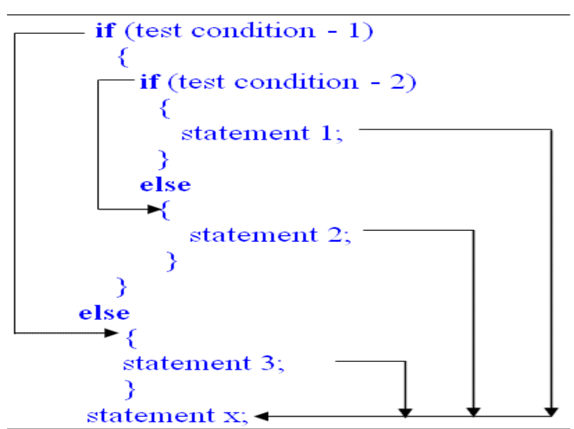
6.3 NESTING OF IF ELSE

When an if else statement is present inside the body of another “if” or “else” then this is called nested if else.

Syntax

```
if(condition) {
//Nested if else inside the body of "if"
if(condition2) {
//Statements inside the body of nested "if
}
else {
/Statements inside the body of nested "else"
}
}
else {
//Statements inside the body of "else"
}
```

Flow Chart



Example:

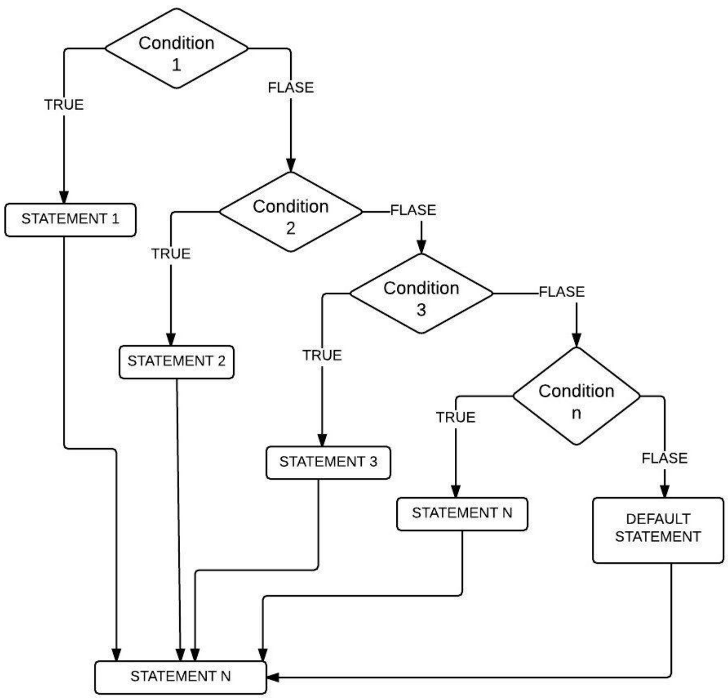
```
#include <stdio.h>
int main()
{
 int var1, var2;
printf("Input the value of var1:");
scanf("%d", &var1);
 printf("Input the value of var2:");
 scanf("%d",&var2);
if (var1 != var2)
{
printf("var1 is not equal to var2\n");
//Nested if else
if (var1 > var2)
{
		printf("var1 is greater than var2\n");
}
else
{
		printf("var2 is greater than var1\n");
}
}
 else
{
printf("var1 is equal to var2\n");
}
return 0;
}
```

6.4 ELSE IF

Use the else if statement to specify a new condition if the first condition is false.

Syntax:

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```



Example:

```
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 20) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

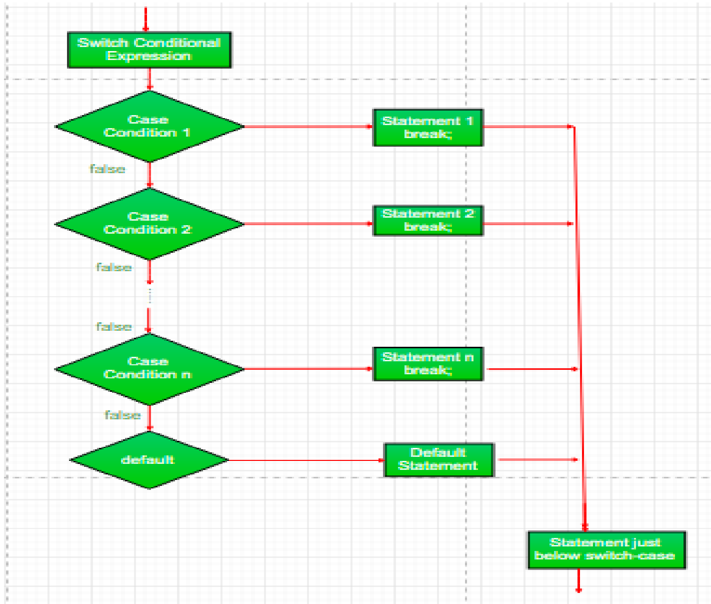
6.5 SWITCH

The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax

```
switch (expression)
{
case value1:
statement1;
 break;
 case value2:
  statement2;
break;
 .
.
 case valueN:
 statementN;
break;
default:
statementDefault;
}
```

Flow Chart



Example:

```
class SwitchCaseDemo
{
 public static void main(String args[])
{
int i = 9;
switch (i)
{
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
default:
System.out.println("i is greater than 2.");
}
}
}
```

6.6 ?: OPERATOR

Java language has an unusual operator, useful for making two way decisions. This operator is a combination of ? and :

Syntax

Conditional1 expression ? expression1 : expression2

Example

```
name.equals("Rumplestiltskin")
? System.out.println("Give back child")
: System.out.println("Laugh");
```

6.7 WHILE.….DO

The while loop loops through a block of code as long as a specified condition is true:

Syntax

```
while (condition)
 {
  // code block to be executed
 }
```

Example

```
int i = 0;
while (i < 5) {
System.out.println(i);
 i++;
}
```

Do/While loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do

{
  // code block to be executed
}
while (condition);
```

Example

```
int i = 0;
do {
System.out.println(i);
 i++;
}
while (i < 5);
```

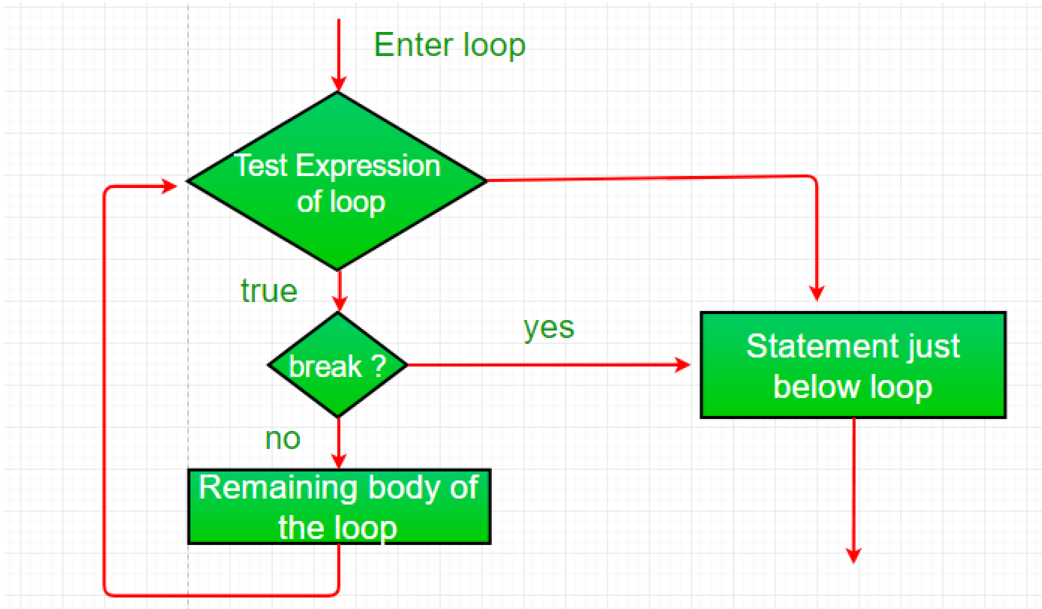
6.8 FOR JUMP IN LOOPS

Java supports three jump statement: **break, continue** and **return**. These three statements transfer control to other part of the program.

1. **Break:** In Java, break is majorly used for:
    - Terminate a sequence in a switch statement (discussed above).
    - To exit a loop.
    - Used as a “civilized” form of goto

    .

Flow Chart:



Example

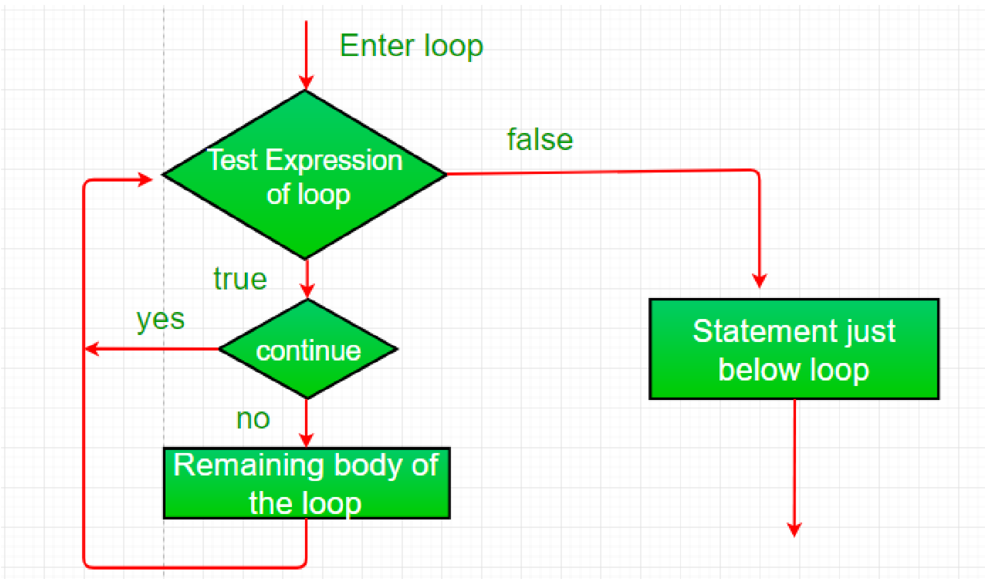
```
class BreakLoopDemo
{
public static void main(String args[])
{
 // Initially loop is set to run from 0-9
for (int i = 0; i < 10; i++)
{
// terminate loop when i is 5.
if (i == 5)
    break;

System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

Continue:

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

Flow Chart:



Example:

```
class ContinueDemo
{
public static void main(String args[])
{
for (int i = 0; i < 10; i++)
{
// If the number is even
// skip and continue
if (i%2 == 0)
    continue;

// If number is odd, print it
System.out.print(i + " ");
  }
 }
}
```

Return:

The return statement is used to explicitly return from a method. That is, it causes a program control to transfer back to the caller of the method.

Example:

```
class Return
{
public static void main(String args[])
{
boolean t = true;
System.out.println("Before the return.");
 if (t)
return;
 // Compiler will bypass every statement
// after return
System.out.println("This won't execute.");
}
}
```

UNIT VII: CLASSES,OBJECTS

7.1 INTRODUCTION

A class, in the context of Java, are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

7.2 ADDING VARIABLES

1. Understand fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries, etc.

2. Be aware of the important topics and principles of software development.

3. Have the ability to write a computer program to solve specified problems.

A class is a blueprint from which individual objects are created.

A class is defined in java using the keyword **class** followed by the name of the class. The body of the class is defined inside the curly brackets and terminated either by a semicolon or list of declaration at the end.

A variable is a name given to a memory location. It is the basic unit of storage in a program.

The value stored in a variable can be changed during program execution.
A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
In Java, all the variables must be declared before use.

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

Example Program

```
public class StudentDetails {
public void StudentRno()
{
// local variable age
 int rno = 0;
 rno = age + 1005;
System.out.println("Student Rno is : " + rno);
}
 public static void main(String args[])
{
StudentDetails obj = new StudentDetails();
obj.StudentRno();
}
}
```

Output:

```
Student Rno is : 1005
```

In the above program, the variable age is a local variable to the function StudentRno(). If we use the variable age outside StudentRno() function, the compiler will produce an error as shown in below program.

7.3 METHODS

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python.
Methods are time savers and help us to reuse the code without retyping the code.

Method Declaration

In general, method declarations has six components :

Modifier-: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.

public: accessible in all class in your application.
protected: accessible within the class in which it is defined and in its **subclass(es)**
private: accessible only within the class in which it is defined.

default (declared/defined without using any modifier) : accessible within 58same class and package within which its class is defined.

The return type : The data type of the value returned by the method or void if does not return a value.
Method Name : the rules for field names apply to method names as well, but the convention is a little different.
Parameter list : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
Exception list : The exceptions you expect by the method can throw, you can specify these exception(s).
Method body : it is enclosed between braces. The code you need to be executed to perform your intended operations.

Types of Java methods

Depending on whether a method is defined by the user, or available in standard library, there are two types of methods:

- Standard Library Methods
- User-defined Methods

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use.These standard libraries come along with the Java Class Library (JCL) in a Java archive(*.jar) file with JVM and JRE.

User-defined Method

You can also define methods inside a class as per your wish. Such methods are called user-defined methods.

```
public static void myprogramme() {

System.out.println("Welcome Java Program"); }
```

7.4 CREATING OBJECT

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cats, humans, etc. All these objects have a state and a behavior.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example:

```
public class sample
{
  int x= 5;
  public static void main(String[]args)
{
        sample ob = new sample();
        System.out.println(ob.x);
        }
}
```

Output:

5

7.5 ACCESSING MEMBERS

The components of a class, such as its instance variables or methods are called the members of a class or class members. A class member is declared with an access modifier to specify how it is accessed by the other classes in Java. A Java class member can take any of the access modifiers, such as - public, protected, default and private.

Modifier	**Class**	**constructor**	**method**	**Data/variables**
Public	Yes	Yes	Yes	Yes
Protected		Yes	Yes	Yes
Default	Yes	Yes	Yes	Yes
Private		Yes	Yes	Yes
Static			Yes	
Final	Yes		Yes	

7.6 CONSTRUCTORS

A **constructor** is a special method that is used to initialize a newly created object and is called just after the memory is allocated for the object. It can be used to initialize the objects to desired values or default values at the time of object creation. It is not mandatory for the coder to write a constructor for a class.

Rules for writing Constructor:

- Constructor(s) of a class must has same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static and Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of constructor

There are two type of constructor in Java:

1. **No-argument constructor:** A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor. Default constructor provides the default values to the object like 0, null, etc. depending on the type.

Example program:

```
class demo
{
     int  v1;
        demo()
    {
     v1 = 10;

    System.out.println("Inside Constructor");
    }

 public void display(){
    System.out.println("V1 ="+v1);

 }

    public static void main(String args[]){
    demo d1 = new Demo();
   d1.display();
 }
}
Inside Constructor
V1 =10
```

7.7 METHOD OVERLOADING

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java that allows a class to have more than one constructor having different argument lists.

Example Program:

```
public class Sum {

// Overloaded sum(). This sum takes two int parameters
public int sum(int x, int y)
{
```

```
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

30
60
31.0

7.8 NESTING OF METHODS

Method within method in java

Java does not support **"directly"** nested methods. Many functional programming languages support method within method. But you can achieve nested method functionality in Java 7 or older version by define local classes, class within method so this does compile. And in java 8 and newer version you achieve it by lambda expression. Let's see how it achieve.

Method 1 (Using anonymous subclasses)

It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class. for more in detail about anonymous inner class

```
// Java program implements method inside method

    public class GFG {
```

```
// create a local interface with one abstract
// method run()
interface myInterface {
 void run();
}
// function have implements another function run()
static void Foo()
{
// implement run method inside Foo() function
  myInterface r = new myInterface() {
 public void run()
{
  System.out.println("geeksforgeeks");
};
};
r.run();
}
public static void main(String[] args)
{
Foo();
}
}
```

Output:

```
geeksforgeeks
```

Method 2 (Using local classes)

You can also implement a method inside a local class. A class created inside a method is called local inner class. If you want to invoke the methods of local inner class, you must instantiate this class inside method.

```
// Java program implements method inside method

public class GFG {

// function have implementation of another

// function inside local class

static void Foo()

{

 // local class

class Local {

void fun()

    {

       System.out.println("geeksforgeeks");

    }

 }

 new Local().fun();

}

  public static void main(String[] args)

{

  Foo();

 }

}
```

Output:

```
geeksforgeeks
```

Method 3 (Using a lambda expression)

Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda

expressions implement the only abstract function and therefore implement functional interfaces. for more about expression

```
// Java program implements method inside method
public class GFG {
interface myInterface {
  void run();
}
 // function have implements another function
// run() using Lambda expression
static void Foo()
{
  // Lambda expression
  myInterface r = () ->
  {
     System.out.println("geeksforgeeks");
  };
  r.run();
 }
  public static void main(String[] args)
 {
  Foo();
 }
}
```

Output:

```
Geeksforgeeks
```

7.9 INHERITANCE

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:

The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Reusablity:

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Inheritance allows us to reuse of code, it improves reusability in your java application.

Note: The biggest advantage of Inheritance is that the code that is already present in base class need not be rewritten in the child class.

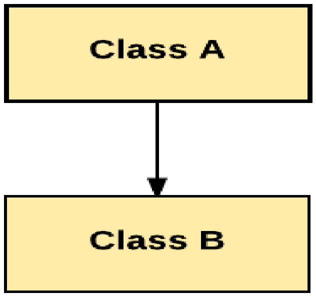
The important point to note in the above example is that the child class is able to access the private members of parent class through protected methods of parent class. When we make a instance variable(data member) or method protected**,** this means that they are accessible only in the class itself and in child class. These public, protected, private etc. are all access specifiers and we will discuss them in the coming tutorials.

Types of inheritance

To learn types of inheritance in detail, refer: Types of Inheritance in Java.

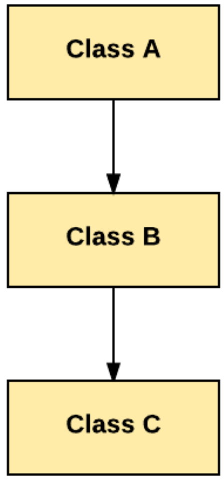
Single Inheritance:

refers to a child and parent class relationship where a class extends the another class.



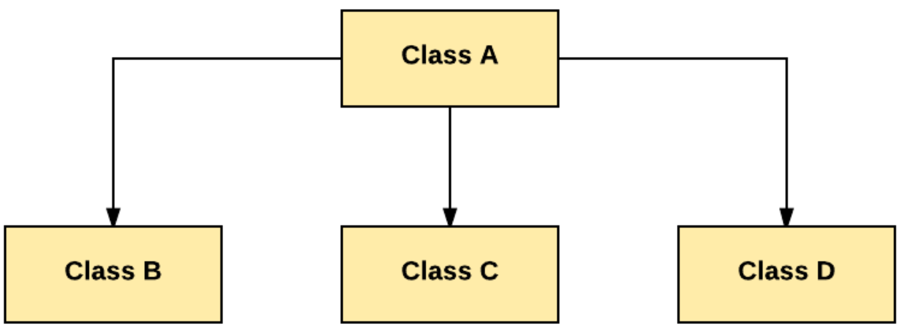
Multilevel inheritance:

refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.



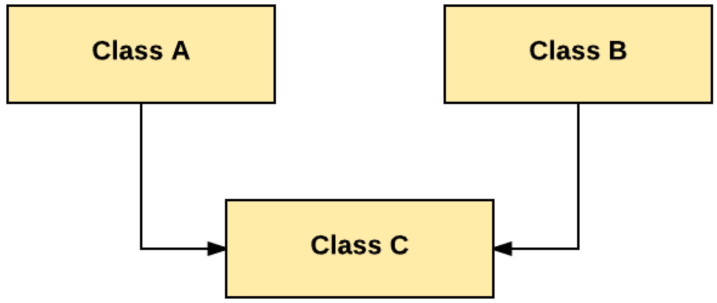
Hierarchical inheritance:

refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.



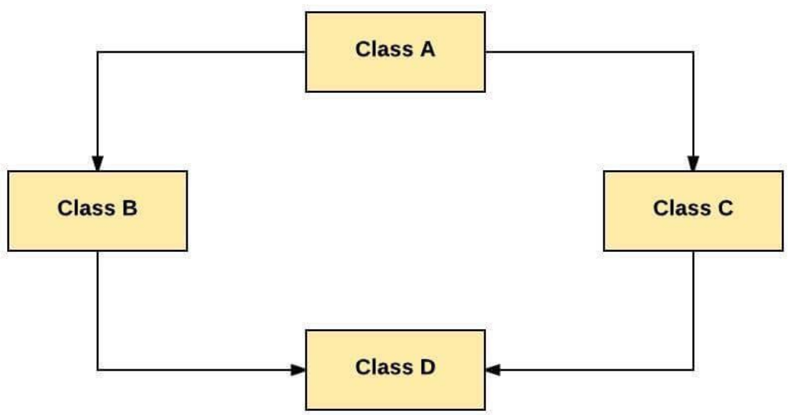
Multiple Inheritance:

refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B.



Hybrid inheritance:

Combination of more than one types of inheritance in a single program. For example class A & B extends class C and another class D extends class A then this is a hybrid inheritance example because it is a combination of single and hierarchical inheritance.



Example:

```
 // base class
class Bicycle
{
// the Bicycle class has two fields
 public int gear;
 public int speed;

// the Bicycle class has one constructor
public Bicycle(int gear, int speed)
```

```
 {
    this.gear = gear;
    this.speed = speed;
 }

 // the Bicycle class has three methods
 public void applyBrake(int decrement)
{
    speed -= decrement;
}

public void speedUp(int increment)
{
    speed += increment;
}
    // toString() method to print info of Bicycle
public String toString()
{
    return("No of gears are "+gear
            +"\n"
            + "speed of bicycle is "+speed);
}
}
```

// derived class

```
  class MountainBike extends Bicycle
{

   // the MountainBike subclass adds one more field
   public int seatHeight;

   // the MountainBike subclass has one constructor
   public MountainBike(int gear,int speed,
                       int startHeight)
   {
      // invoking base-class(Bicycle) constructor
      super(gear, speed);
      seatHeight = startHeight;
   }

   // the MountainBike subclass adds one more method
   public void setHeight(int newValue)
   {
      seatHeight = newValue;
   }

   // overriding toString() method
   // of Bicycle to print more info
```

```
    @Override
    public String toString()
    {
        return (super.toString()+
                "\nseat height is "+seatHeight);
    }

  }

// driver class
public class Test
{
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());

    }
}
```

Output:

```
No of gears are 3

speed of bicycle is 100

seat height is 25
```

7.10 OVERRIDING METHODS

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. Rules for method overriding:

- In java, a method can only be written in Subclass, not in same class.
- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level.

- For example: if the super class method is declared public then the over-ridding method in the sub class cannot be either private or protected.

- A method declared static cannot be overridden but can be re-declared.
- Constructors cannot be overridden.

Sample Example:

```
package com.crunchify.tutorials;

public class CrunchifyObjectOverriding {

public static void main(String args[]) {

Company a = new Company(); // Company reference and object

Company b = new eBay(); // Company reference but eBay object

a.address();// runs the method in Company class

b.address();// Runs the method in eBay class

}

}

class Company {

public void address() {

System.out.println("This is Address of Crunchify Company...");

}

}

class eBay extends Company {

public void address() {

        System.out.println("This is eBay's Address...");

    }

}
```

Output:

```
This is Address of Crunchify Company...

This is eBay Rock's Address...
```

7.11 FINAL CLASS

Inheritance is surely one of the highly useful features in Java. But at times, it may be desired that a class should not be extendable by other classes to prevent exploitation. For such purpose, we have the final keyword. We have already seen even what final variables are. Final classes and methods are also similar. A class declared as final cannot be extended while a method declared as final cannot be overridden in its subclasses. A method or a class is declared to be final using the final keyword. Though a final class cannot be extended, it can extend other classes. In simpler words, a final class can be a sub class but not a super class.

```
final public class A {
   //code
}
```

The final keyword can be placed either before or after the access specifier. The following declaration of class A is equivalent to the above.

```
public final class A {
   //code
}
```

Final methods are also declared in a similar way. Here too, there should be no specifiers stated between the return type and the method name. For example, to make a static method declared with the public access specifier final, the three specifiers, public, static and final may be placed in any order to the left of the return type.

```
public final void someMethod() {
   //code
}
```

When we attempt to extend a final class or override a final method, compilation errors occur.

```
class B extends A { // compilation error, A is final
}
```

We will now look into a small trivial example to understand the necessity of final classes and methods. Consider that you are a developer writing classes for use by others. You have two classes named Car and Vehicle defined in the following way and of course with may other methods of your own.

```
public class Car {

    public void move() {
    System.out.println("Moving on road");
    }
    }
   class MoveAVehicle {
   public static void move(Car c) {
    c.move();
   }
 }
```

Now, since the class Car is not final, other people may extend it to their own classes. Following is an example:

```
    class Aeroplane extends Car {
    public void move() {
    System.out.println("Moving in air");
 }
}
```

UNIT VIII: ARRAYS, STRINGS AND VECTORS

8.1 ARRAYS

Introduction arrays

An array is a collection of similar data types. Array is a container object that hold values of homogenous type. It is also known as static data structure because size of an array must be specified at the time of its declaration.

An array can be either primitive or reference type. It gets memory in heap area. Index of array starts from zero to size-1.

Objectives array

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Array features

- It is always indexed. Index begins from 0.
- It is a collection of similar data types.
- It occupies a contiguous memory location.

An array declaration has two components: the type and the name. sp

type declares the element type of the array. The element type determines the data type of each element that comprises the array. Like array of int type, we can also create an array of other

primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.

Syntax :

Datatype[]identifier;

Or

Datatype identifier[];

8.2 ONE DIMENSIONAL ARRAY

One-dimension means that there is only one parameter to deal with. In regular terms, it is the length of something. Similarly, as far as an array is concerned, one dimension means it has only one value per location or index.

One-dimensional array in Java programming is an array with a bunch of values having been declared with a single index.

Example:

```
class OnedimensionalLoop

{

 public static void main(String args[])

   {

 int a[]={10,20,30,40,50};//declaration and initialization

System.out.println("One dimensional array elements are :\n");

 for(int i=0;i<a.length;i++)

{

 System.out.println("a["+i+"]:"+a[i]);

     }

   }}
```

Output:

One dimensional array elements are :

a[0]:10

a[1]:20

a[2]:30

a[3]:40

a[4]:50

8.3 TWO DIMENSIONAL ARRAY

Two – dimensional array is the simplest form of a multidimensional array. A two – dimensional array can be seen as an array of one – dimensional array for easier understanding.

Indirect Method of Declaration:

Syntax:

```
data_type[][] array_name = new data_type[x][y];

For example: int[][] arr = new int[10][20];
```

Representation of 2D array in Tabular Format:

A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Example:

filter_none
edit
play_arrow
brightness_4

```
class GFG {
public static void main(String[] args)
{
  int[][] arr = { { 1, 2 }, { 3, 4 } };
  for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 2; j++) {
        System.out.print(arr[i][j] + " ");
    }
      System.out.println();
      }
      }
   }
```

Output:

```
1 2

3 4
```

8.4 STRING

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write −

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

Example

```
public class StringDemo {
public static void main(String args[]) {
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println( helloString );
 }
}
```

This will produce the following result −

Output

hello.

Java String Methods

Here are the list of the methods available in the Java String class. These methods are explained in the separate tutorials with the help of examples. Links to the tutorials are provided below:

1. **char charAt(int index):** It returns the character at the specified index. Specified index value should be between 0 to length() -1 both inclusive. It throws IndexOutOfBoundsException if index<0||>= length of String.
2. **boolean equals(Object obj):** Compares the string with the specified string and returns true if both matches else false.
3. **boolean equalsIgnoreCase(String string):** It works same as equals method but it doesn't consider the case while comparing strings. It does a case insensitive comparison.
4. **int compareTo(String string):** This method compares the two strings based on the Unicode value of each character in the strings.
5. **int compareToIgnoreCase(String string):** Same as CompareTo method however it ignores the case during comparison.
6. **boolean startsWith(String prefix, int offset):** It checks whether the substring (starting from the specified offset index) is having the specified prefix or not.
7. **boolean startsWith(String prefix):** It tests whether the string is having specified prefix, if yes then it returns true else false.
8. **boolean endsWith(String suffix):** Checks whether the string ends with the specified suffix.
9. **int hashCode():** It returns the hash code of the string.
10. **int indexOf(int ch)**: Returns the index of first occurrence of the specified character ch in the string.
11. **int indexOf(int ch, int fromIndex)**: Same as indexOf method however it starts searching in the string from the specified fromIndex.
12. **int lastIndexOf(int ch):** It returns the last occurrence of the character ch in the string.
13. **int lastIndexOf(int ch, int fromIndex):** Same as lastIndexOf(int ch) method, it starts

search from fromIndex.

14. **int indexOf(String str):** This method returns the index of first occurrence of specified substring str.
15. **int lastindexOf(String str):** Returns the index of last occurrence of string str.
16. **String substring(int beginIndex):** It returns the substring of the string. The substring starts with the character at the specified index.
17. **String substring(int beginIndex, int endIndex)**: Returns the substring. The substring starts with character at beginIndex and ends with the character at endIndex.
18. **String concat(String str):** Concatenates the specified string "str" at the end of the string.
19. **String replace(char oldChar, char newChar):** It returns the new updated string after changing all the occurrences of oldChar with the newChar.
20. **boolean contains(CharSequences):** It checks whether the string contains the specified sequence of char values. If yes then it returns true else false. It throws NullPointerException of 's' is null.
21. **String toUpperCase(Locale locale)**: Converts the string to upper case string using the rules defined by specified locale.
22. **String toUpperCase():** Equivalent to toUpperCase(Locale.getDefault()).
23. **public String intern():** This method searches the specified string in the memory pool and if it is found then it returns the reference of it, else it allocates the memory space to the specified string and assign the reference to it.
24. **public boolean isEmpty()**: This method returns true if the given string has 0 length. If the length of the specified Java String is non-zero then it returns false.
25. **public static String join()**: This method joins the given strings using the specified delimiter and returns the concatenated Java String
26. **String replaceFirst(String regex, String replacement):** It replaces the first occurrence of substring that fits the given regular expression "regex" with the specified replacement string.
27. **String replaceAll(String regex, String replacement)**: It replaces all the occurrences of substrings that fits the regular expression regex with the replacement string.
28. **String[] split(String regex, int limit):** It splits the string and returns the array of substrings that matches the given regular expression. limit is a result threshold here.
29. **String[] split(String regex):** Same as split(String regex, int limit) method however it does not have any threshold limit.
30. **String toLowerCase(Locale locale):** It converts the string to lower case string using the rules defined by given locale.
31. **public static String format():** This method returns a formatted java String
32. **String toLowerCase():** Equivalent to toLowerCase(Locale. getDefault()).
33. **String trim():** Returns the substring after omitting leading and trailing white spaces from the original string.
34. **char[] toCharArray():** Converts the string to a character array.
35. **static String copyValueOf(char[] data):** It returns a string that contains the characters of the specified character array.
36. **static String copyValueOf(char[] data, int offset, int count)**: Same as above method with two extra arguments – initial offset of subarray and length of subarray.
37. **void getChars(int srcBegin, int srcEnd, char[] dest, int destBegin):** It copies the characters of **src**array to the **dest** array. Only the specified range is being copied(srcBegin to srcEnd) to the dest subarray(starting fromdestBegin).

38. **static String valueOf()**: This method returns a string representation of passed arguments such as int, long, float, double, char and char array.
39. **boolean contentEquals(StringBuffer sb):** It compares the string to the specified string buffer.
40. **boolean regionMatches(int srcoffset, String dest, int destoffset, int len):** It compares the substring of input to the substring of specified string.
41. **boolean regionMatches(boolean ignoreCase, int srcoffset, String dest, int destoffset, int len):** Another variation of regionMatches method with the extra boolean argument to specify whether the comparison is case sensitive or case insensitive.
42. **byte[] getBytes(String charsetName):** It converts the String into sequence of bytes using the specified charset encoding and returns the array of resulted bytes.
43. **byte[] getBytes():** This method is similar to the above method it just uses the default charset encoding for converting the string into sequence of bytes.
44. **int length():** It returns the length of a String.
45. **boolean matches(String regex):** It checks whether the String is matching with the specified regular expression regex.
46. **int codePointAt(int index):**It is similar to the charAt method however it returns the Unicode code point value of specified index rather than the character itself.

8.5 VECTOR

The Vector class implements a growable array of objects. Vectors basically falls in legacy classes but now it is fully compatible with collections.

- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
- It extends **AbstractList** and implements **List** interfaces.

`Vector` implements List Interface. Like `ArrayList` it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

Three ways to create vector class object:

Method 1:

```
Vector vec = new Vector();
```

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

ArrayList and Vector both implements List interface and maintains insertion order.

However, there are many differences between ArrayList and Vector classes that are given

below.

ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the **Iterator** interface to traverse the elements.	A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

8.6 WRAPPER CLASSES

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive**.**

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the ***java.lang*** package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
Boolean	Boolean
Char	Character
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
Double	Double

Autoboxing and Unboxing:

Autoboxing: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

Example:

filter_none

edit
play_arrow
brightness_4

```
// Java program to demonstrate Autoboxing
import java.util.ArrayList;
class Autoboxing
{
public static void main(String[] args)
{
char ch = 'a';
// Autoboxing- primitive to Character object conversion
Character a = ch;
ArrayList<Integer> arrayList = new ArrayList<Integer>();
// Autoboxing because ArrayList stores only objects
arrayList.add(25);
// printing the values from object
System.out.println(arrayList.get(0));
}
}
```

Output:

```
25
```

Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.

filter_none
edit
play_arrow
brightness_4

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;
class Unboxing
{
public static void main(String[] args)
{
Character ch = 'a';
// unboxing - Character object to primitive conversion
char a = ch;
ArrayList<Integer> arrayList = new ArrayList<Integer>();
arrayList.add(24);
// unboxing because get method returns an Integer object
int num = arrayList.get(0);
// printing the values from primitive data types
System.out.println(num);
```

```
    }
}
```

Output:

```
2
```

UNIT XI: INTERFACES

INTERFACES

Introduction

In this article we will discuss about interface in JAVA. As multiple inheritance in JAVA only achieve through **Inerface**. So we discuss about **Interface in JAVA.**

- In computing, interface is act like a communicator between peripheral devices(monitor, keyboard) and computer system.
- In field of **computer science**, an interface refers as a interactor between components.
- In computer any type of communication of computer system with user we say it is an interface.

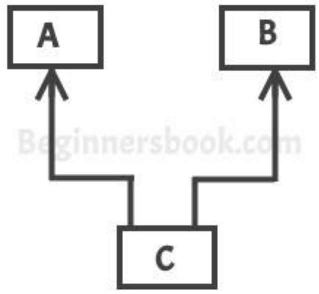
Objective

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
  The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

9.1 MULTIPLE INHERITANCE

When one class extends more than one classes then this is called **multiple inheritance**. For example: Class C extends class A and B then this type of inheritance is known as multiple inheritance. Java doesn't allow multiple inheritance. In this article, we will discuss why java

doesn't allow multiple inheritance and how we can use interfaces instead of classes to achieve the same purpose.

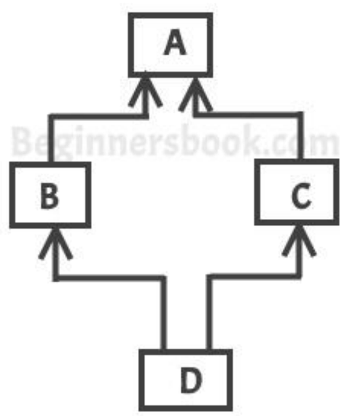


Multiple Inheritance

1.2 Why Java doesn't support multiple inheritance?

C++ , Common lisp and few other languages supports multiple inheritance while java doesn't support it. Java doesn't allow multiple inheritance to **avoid the ambiguity** caused by it. One of the example of such problem is the **diamond problem** that occurs in multiple inheritance.

To understand the basics of inheritance, refer this main guide: Inheritance in Java



Hybrid Inheritance

1.3 Can we implement more than one interfaces in a class

Yes, we can implement more than one interfaces in our program because that doesn't cause any ambiguity(see the explanation below).

```
interface X
{
  public void myMethod();
}
interface Y
{
  public void myMethod();
}
class JavaExample implements X, Y
{
  public void myMethod()
  {
     System.out.println("Implementing more than one interfaces");
  }
  public static void main(String args[]){
            JavaExample obj = new JavaExample();
            obj.myMethod();
  }
}
```

Output:

```
Implementing more than one interfaces
```

9.2 DEFINING INTERFACE

.An interface is basically kind of class. Like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields

Syntax:

```
Interface InterfaceName

{

Variables declaration;

Methods declaration;
```

}

Here, interface is the key word and InterfaceName is any valid java variables.

Static final type VariableName = Value;

Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements

Return-type methodName1 (Parameter_list);

Example:

```
Interface Item

{

Static final int code=1001;

Static final string name = "Fan";

Void display();

}
```

9.3 EXTENDING INTERFACE

Like classes, interfaces can also be extended. That is, an interface can be subinterface from other interfaces. This is achieved using the keyword **extends**.

syntax

```
Interface name2 extends name1

{

Body of name2

}
```

Example:

```
Interface ItemConstants

{

Int code = 1001;

String  name = "Fan";
```

```
}

Interface Item extends Itemconstants

{

Void display();

}
```

9.4 IMPLEMENTING INTERFACES

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always abstract by default, which doesn't let them give their implementation (or method definition) in interface itself

To declare a class that implements an interface, you include an implements clause in the class declaration. Your class can implement more than one interface, so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the `implements` clause follows the extends clause, if there is one.

Syntax

```
Class classname implements interfacename

{

Body of classname

}
```

Any class can implement `Relatable` if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice (see the RectanglePlus class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the isLargerThan`()` method.

If you know that a class implements `Relatable`, then you know that you can compare the size of the objects instantiated from that class.

1.4 Implementing the Relatable Interface

Here is the Rectangle class that was presented in the Creating Objects section, rewritten to implement Relatable.

```
public class RectanglePlus
```

```
implements Relatable {
public int width = 0;
public int height = 0;
public Point origin;
// four constructors
public RectanglePlus() {
   origin = new Point(0, 0);
}
public RectanglePlus(Point p) {
   origin = p;
}
public RectanglePlus(int w, int h) {
   origin = new Point(0, 0);
   width = w;
   height = h;
}
public RectanglePlus(Point p, int w, int h) {
   origin = p;
   width = w;
   height = h;
}

// a method for moving the rectangle
public void move(int x, int y) {
   origin.x = x;
   origin.y = y;
}

// a method for computing
// the area of the rectangle
public int getArea() {
   return width * height;
}

// a method required to implement
// the Relatable interface
public int isLargerThan(Relatable other) {
   RectanglePlus otherRect
      = (RectanglePlus)other;
   if (this.getArea() < otherRect.getArea())
      return -1;
   else if (this.getArea() > otherRect  RectanglePlus .getArea())
      return 1;
     else
      return 0;
    }
   }
```

Because RectanglePlus implements Relatable, the size of any two objects can be compared.

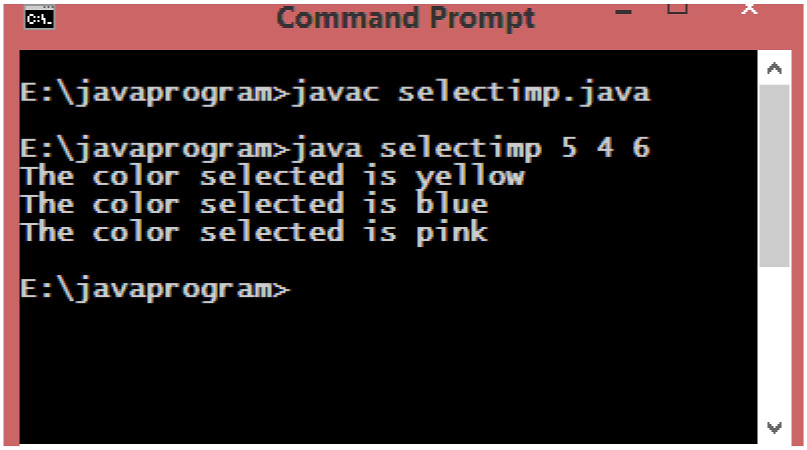
9.5 ACCESSING INTERFACE VARIABLES

A Java interface can contain both variables and constants. However, often it does not makes sense to place variables in an interface. In some cases it can make sense to define constants in an interface. Especially if those constants are to be used by the classes implementing the interface, e.g. in calculations, or as parameters to some of the methods in the interface. However, my advice to you is to avoid placing variables in Java interfaces if you can.

All variables in an interface are public, even if you leave out the public keyword in the variable declaration.

```
interface selectcolor
{
int blue=4;
int yellow=5;
int pink=6;
public void choose(int color);
}
class selectimp implements selectcolor
{
public void choose(int color)
{
switch(color)
{
case blue:
  System.out.println("The color selected is blue");
  break;
case yellow:
  System.out.println("The color selected is yellow");
  break;
case pink:
  System.out.println("The color selected is pink");
  break;
}
}
public static void main(String aa[])
{
int a1,b1,c1;
a1=Integer.parseInt(aa[0]);
b1=Integer.parseInt(aa[1]);
c1=Integer.parseInt(aa[2]);
selectimp st=new selectimp();
st.choose(a1);
st.choose(b1);
st.choose(c1);
```

```
}
}
```



Explanation....

In this example first an interface selectcolor is created and the value for the integers blue, yellow, pink are set as 4,5,6.

Then a method choose() which takes an integer parameter is declared.

A class selectimp is created, which implements the interface selectcolor.

Then the method choose() of the interface selectcolor is implemented using the switch case statements.

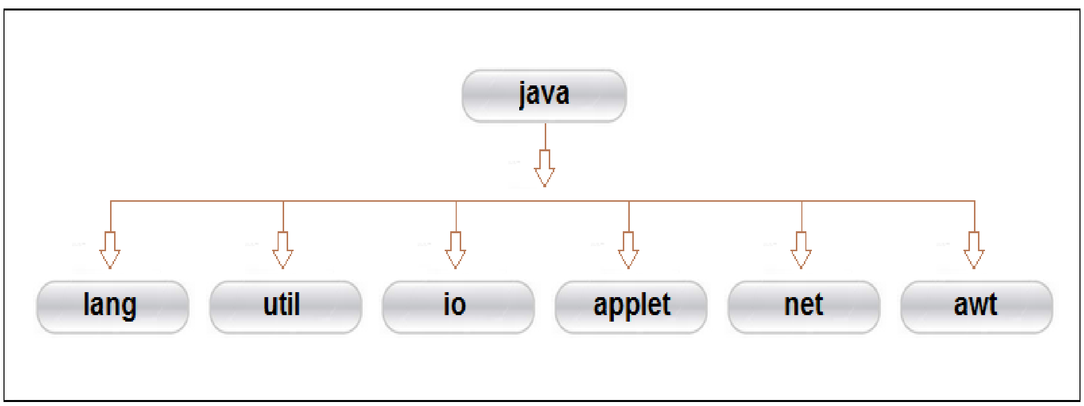
Then there is a main() which creates the object of the class selectimp and call the choose() of the selectimp class with different parameters or arguments.

UNIT X: API PACKAGE

10.1 USING PACKAGES

Java system APl(Application Program Interface) **p**rovides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the the Java API. Following figure shows the system packages that are frequently used in the programs

.



java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of

	data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

The package are organized in a hierarchical structure, the package named java contains the package awt, which in turn contains various classes required for implementing graphical user interface.

Syntax:

Import packagename.classname;

Or

Import packagename.*

10.2 NAMING CONVENTIONS

Below are some naming conventions of java programming language. They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constant

Classes and Interfaces :

Class names should be **nouns**, in mixed case with the **first** letter of each internal word capitalised. Interfaces name should also be capitalised just like class names.

Use whole words and must avoid acronyms and abbreviations

Examples:

```
interface  Bicycle
class MountainBike implements Bicyle


interface Sport
```

```
class Football implements Sport
```

Methods :

Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalised.

Examples:

```
void changeGear(int newValue);

void speedUp(int increment);

void applyBrakes(int decrement);
```

Variables:

Variable names should be short yet meaningful.

- Should not start with underscore('_') or dollar sign '$' characters.
- Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.
- One-character variable names should be avoided except for temporary variables.
- Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

Examples:

```
// variables for MountainBike class

   int speed = 0;

   int gear = 1;
```

Constant variables:

Should be **all uppercase** with words separated by underscores ("_").
There are various constants used in predefined classes like Float, Long, String etc.

Examples:

```
static final int MIN_WIDTH = 4;


// Some  Constant variables used in predefined Float class

public static final float POSITIVE_INFINITY = 1.0f / 0.0f;

public static final float NEGATIVE_INFINITY = -1.0f / 0.0f;

public static final float NaN = 0.0f / 0.0f;
```

Packages:

- The prefix of a unique package name is always written in **all-lowercase ASCII letters** and should be one of the top-level domain names, like com, edu, gov, mil, net, org.
- Subsequent components of the package name vary according to an organisation's own internal naming conventions.

Examples:

```
com.sun.eng

com.apple.quicktime.v2


// java.lang packet in JDK

java.lang
```

10.3 CREATING PACKAGES

A Package is a collection of related classes. It helps organize your classes into a folder structure and make it easy to locate and use them. More importantly, it helps improve re-usability.Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

Syntax:-

```
        package nameOfPackage;
```

- **java.lang** − bundles the fundamental classes
- **java.io** − classes for input , output functions are bundled in this package

Example

Following package example contains interface named *animals* −

```
/* File name : Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

Now, let us implement the above interface in the same package *animals* −

```
package animals;
/* File name : MammalInt.java */
```

```
public class MammalInt implements Animal {

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

Now compile the java files as shown below −

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.

Output

You can execute the class file within the package and get the result as shown below.

```
Mammal eats
Mammal travels
```

10.4 ACCESSING PACKAGES

It may be recalled that we have discussed earlier that a java system package can be accessed either using fully qualified class name or using a shortcut through the import statement.

Syntax

Import package1 [.package2] [package3].classname;

Here package1 is the name of the top level package, package2 is the name of the package that is inside package1,and soon

Import firstPackage.secondpackage.MyClass;

10.5 USING A PACKAGE

Let us now consider some simple programs that will use classes from other packages. The listing below shows a package named package1 containing a single class ClassA.

```
Package package1;

Public class classA;

{ Public void displayA()

{

System.out.println ("class A");

}

}
```

This source file should be named Class A.java and stored in the subdirectory package1 as stated earlier. Now compile this java file. The resultant Class A.class will be stored in the same subdirectory.

```
import package1.ClassA;

class packageTest1

{

public static void main (String args[])

{

ClassA objectA=new ClassA();

objectA.displayA();

}

}
```

10.6 ADDING A CLASS TO PACKAGE

It is simple to add a class to an existing package.

Package p1;

Public classA;

{ //body of A

}

The package p1 contains one public class by name A. Suppose we want to add another class B to this package.

Define the class and make it public

Place the package statement

Package p1;

Before the class definition as follows;

Package pl

Public class B

{ //body of B

}

Store this as B.java file under the directory P1.

Compile B..java file. This will create a B.class file and place it in the directory p1. Note that we can also add a non public class to a package using the same procedure. Now, the package P1 will contain both the classes A and B.

Import p1.*;

UNIT XI: BASICS

11.1 CREATING THREADING

Java defines two ways by which a thread can be created

- By implementing the **Runnable** interface.
- By extending the **Thread** class.
- run() method introduces a concurrent thread into your program. This thread will end when run() method terminates.
- You must specify the code that your thread will execute inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

Syntax

```
Public void run()
```

Example

```
class MyThread implements Runnable
{
public void run()
    {
    System.out.println("concurrent thread started running..");
    }}
```

```
class MyThreadDemo
{
 public static void main(String args[])
        {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
        }}
```

11.2 EXTENDING THE THREAD CLASS

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override **run()** method which is the entry point of new thread.

Example:

```
class ExtendingThread extends Thread
{
 String s[]={"Welcome","to","Java","Programming","Language"};
 public static void main(String args[])
   {
  ExtendingThread t=new ExtendingThread("Extending Thread Class");
   }
  public ExtendingThread (String n)
  {
       super(n);
       start();
  }
     public void run()
  {
        String name=getName();
        for(int i=0;i<s.length;i++)
            {
           try
            {
             sleep(500);
            }
             catch(Exception e)
                {
                }
                 System.out.println(name+":"+s[i]);
     }
```

```
    }
   }
```

11.3 STOPPING AND BLOCKING A THREAD

Starting a thread:

When we create a thread by taking instance of a thread class, the thread will be in its new-bornstate. To move it to runnable state, we use a method named start( ). When we invoke this method with a thread, java run-time schedules it to run by invoking it run() method. Now the thread will be in its running state. To start a thread we use the following syntax:

```
MyThread t1 = new MyThread();

t1.start();
```

Stopping a thread:

Whenever we want to stop a thread from running further, we may do so by calling its stop() method. This causes a thread to stop immediately and move it to its dead state. It forces the thread to stop abruptly before its completion i.e. it causes premature death. To stop a thread we use the following syntax:

```
t1.stop();
```

Blocking a Thread:

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

```
sleep(t) // blocked for ‘t’ milliseconds

suspend() // blocked until resume() method is invoked

wait() // blocked until notify () is invoked
```

These methods cause the thread to go into the *blocked* (or *not-runnable*) state. The thread will return to the runnable state when the specified time is elapsed in the case of sleep( ), the resume() method is invoked in the case of suspend( ), and the notify( ) method is called in the case of wait()

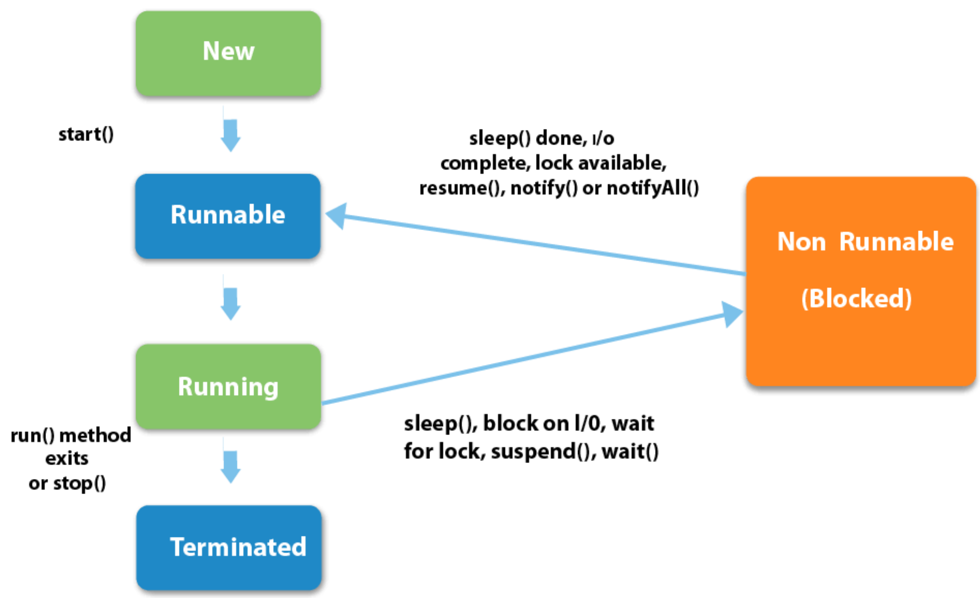
11.4 LIFE CYCLE OF A THREAD

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



New:

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

Runnable:

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

Running:

The thread is in running state if the thread scheduler has selected it.

Non-Runnable (Blocked):

This is the state when the thread is still alive, but is currently not eligible to run.

Terminated:

A thread is in terminated or dead state when its run() method exits.

Example:

```
class thread implements Runnable
{
public void run()
{
// moving thread2 to timed waiting state
 try
{
   Thread.sleep(1500);
}
catch (InterruptedException e)
{
   e.printStackTrace();
}

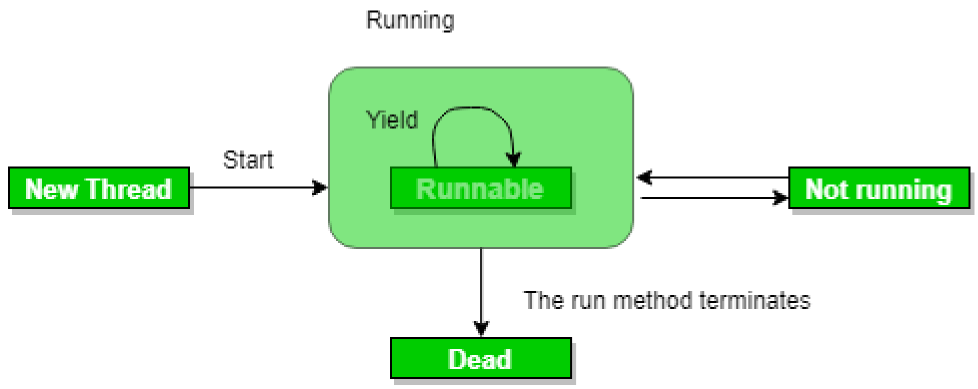
System.out.println("State of thread1 while it called join() method on thread2 -"+
Test.thread1.getState());
try
{
   Thread.sleep(200);
}
catch (InterruptedException e)
{
   e.printStackTrace();
}
}
}
public class Test implements Runnable
{
public static Thread thread1;
public static Test obj;
public static void main(String[] args)
{
obj = new Test();
thread1 = new Thread(obj);
// thread1 created and is currently in the NEW state.
System.out.println("State of thread1 after creating it - " + thread1.getState());
```

```
thread1.start();
 // thread1 moved to Runnable state
 System.out.println("State of thread1 after calling .start() method on it - " +
 thread1.getState());
}
  public void run()
{
 thread myThread = new thread();
 Thread thread2 = new Thread(myThread);
 // thread1 created and is currently in the NEW state.
 System.out.println("State of thread2 after creating it - "+ thread2.getState());
 thread2.start();
// thread2 moved to Runnable state
System.out.println("State of thread2 after calling .start() method on it - " +
thread2.getState());

   // moving thread1 to timed waiting state
    try
   {
    //moving thread1 to timed waiting state
    Thread.sleep(200);
    }
    catch (InterruptedException e)
    {
     e.printStackTrace();
     }
     System.out.println("State of thread2 after calling .sleep() method on it - "+
     thread2.getState() );
     try
    {
     // waiting for thread2 to die
     thread2.join();
    }
   catch (InterruptedException e)
  {
     e.printStackTrace();
  }
  System.out.println("State of thread2 when it has finished it's execution - " +
     thread2.getState());
 }
 }
```

11.5 USING THREAD METHODS

We have discussed how Thread class methods can be used to control the behavior of a thread, We have used the methods start() and run(). There are also methods that can move a thread from one state to another. Yield(), sleep(), and stop() methods.



Use of yield method:

Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.

Syntax:

```
public static native void yield()
```

Yield() indicates that the thread is not doing anything particularly important and if any other threads or processes need to be run, they can. Otherwise, the **current thread will continue to run.**

Example:

```
import java.lang.*;

// MyThread extending Thread
class MyThread extends Thread
{
   public void run()
   {
      for (int i=0; i<5 ; i++)
         System.out.println(Thread.currentThread().getName()
                           + " in control");
   }
}

// Driver Class
public class yieldDemo
```

```
{
   public static void main(String[]args)
   {
      MyThread t = new MyThread();
      t.start();

      for (int i=0; i<5; i++)
      {
         // Control passes to child thread
         Thread.yield();

         // After execution of child Thread
         // main thread takes over
         System.out.println(Thread.currentThread().getName()
                        + " in control");
      }
   }
}
```

sleep():

This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, **the CPU will be idle** (and probably enter a power saving mode).

```
import java.lang.*;

public class SleepDemo implements Runnable
{
   Thread t;
   public void run()
   {
      for (int i = 0; i < 4; i++)
      {
         System.out.println(Thread.currentThread().getName()
                                 + "  " + i);
         try
         {
            // thread to sleep for 1000 milliseconds
            Thread.sleep(1000);
         }

         catch (Exception e)
         {
            System.out.println(e);
         }
      }
```

```
    }

    public static void main(String[] args) throws Exception
    {
        Thread t = new Thread(new SleepDemo());

        // call run() function
        t.start();

        Thread t2 = new Thread(new SleepDemo());

        // call run() function
        t2.start();
    }

}
```

11.6 THREAD EXCEPTIONS

The call to sleep() method is enclosed in a try block and followed by a catch block. This is necessary because the sleep() method throws an exception, which should be caught. If we fail to catch the exception program will not compile.

```
catch (ThreadDeath e)

{

……. //killed thread

}

catch (InterruptedException e)

{

……….. // cannot handle it in the current state

}

catch (Illegal ArgumentException e)

{

…………  //Illegel method argument

}

catch (Exception e)

{
```

………… //any other

}

11.7 SYNCHRONIZATION

Multithreading and synchronization are a very important topic for any Java programmer. Good knowledge of multithreading, synchronization, and thread-safety can put you in front of other developers, at the same time, it's not easy to master this concept. In fact writing correct concurrent code is one of the hardest things, even in Java utilities.

Since Java provides different constructs to provide synchronization and locking e.g. volatilekeyword, atomic variable, explicitly locking using java.util.concurrent.lock.Lock interface and there popular implementations e.g. ReentrantLock and ReentrantReadWriteLock, It becomes even more important to understand difference between synchronized and other constructs

Example:

```
public class Counter{

  private static int count = 0;

  public static synchronized int getCount(){
    return count;
  }

  public synchoronized setCount(int count){
     this.count = count;
  }
}
```

11.8 IMPLEMENTING THE RUNNABLE INTERFACE

- A Thread can be created by extending Thread class also. But Java allows only one class to extend, it wont allow multiple inheritance. So it is always better to create a thread by implementing Runnable interface. Java allows you to impliment multiple interfaces at a time.
- By implementing Runnable interface, you need to provide implementation for run() method.
- To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method.

- Implementing Runnable interface does not create a Thread object, it only defines an entry point for threads in your object. It allows you to pass the object to the Thread(Runnable implementation) constructor.

Example

```
package com.myjava.threads;

class MyRunnableThread implements Runnable{

    public static int myCount = 0;
    public MyRunnableThread(){

    }
    public void run() {
        while(MyRunnableThread.myCount <= 10){
            try{
                System.out.println("Expl Thread: "+(++MyRunnableThread.myCount));
                Thread.sleep(100);
            } catch (InterruptedException iex) {
                System.out.println("Exception in thread: "+iex.getMessage());
            }
        }
    }
}
public class RunMyThread {
    public static void main(String a[]){
        System.out.println("Starting Main Thread...");
        MyRunnableThread mrt = new MyRunnableThread();
        Thread t = new Thread(mrt);
        t.start();
        while(MyRunnableThread.myCount <= 10){
            try{
                System.out.println("Main Thread: "+(++MyRunnableThread.myCount));
                Thread.sleep(100);
            } catch (InterruptedException iex){
                System.out.println("Exception in main thread: "+iex.getMessage());
            }
        }
        System.out.println("End of Main Thread...");
    }
}
```

UNIT XII: MANAGING ERRORS

12.1 TYPES OF ERRORS

Error may broadly be classified into three categories:

- Compile-time errors
- Run time errors
- Logical errors.

Compile-time errors:

These errors are errors which prevents the code from compiling because of error in the syntax such as missing a semicolon at the end of a statement or due to missing braces, class not found, etc. These errors will be detected by java compiler and displays the error onto the screen while compiling.

Run time errors:

These errors are errors which occur when the program is running. Run time errors are not detected by the java compiler. It is the JVM which detects it while the program is running.

Logical errors:

These errors are due to the mistakes made by the programmer. It will not be detected by a compiler nor by the JVM. Errors may be due to wrong idea or concept used by a programmer while coding.

Example:

```
Class error2

{
```

```
Public static void main(String args[])

{

Int a=10;

Int b=5;

Int c=5;

Int x=a/(b-c);

System.out.println("x=" +x);

Int y=a/(b+c);

System.out.println("y="+y);

}

}
```

12.2 EXCEPTION HANDLING CODE:

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Example:

```
public class JavaExceptionExample{
 public static void main(String args[]){
  try{       //code that may raise exception
int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}     //rest code of the program
System.out.println("rest of the code...");
  }
}
```

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
Catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
Finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
Throw	The "throw" keyword is used to throw an exception.
Throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

12.3 MULTIPLE CATCH STATEMENTS

There has always been criticism of checked exceptions in Java exception handling for being verbose and cluttering the code with try-catch blocks.

In Java 7 two new features- try-with-resources (Automatic resource management) and multi-catch statement have been added to mitigate that problem to certain extent.

Syntax:

```
……….
try
{ statement; //generates an exception
}
catch (Exception-Type-1 e)
{
Statement;  //processes exception type1
}
catch (Exception-Type-2 e)
{
Statement; // processes exception type2
}
catch (Exception-Type-N e)
{
Statement ; // processes exception typeN
}
```

Example

```
catch(IOException exp){
   logger.error(exp);
   throw exp;
}catch(SQLException exp){
   logger.error(exp);
   throw exp;
}
```

Java multi-catch - Handling more than one type of exception

Java 7 onward it is possible to catch multiple exceptions in one catch block, which eliminates the duplicated code. Each exception type within the multi-catch statement is separated by Pipe symbol (|).

```
catch(IOException | SQLException exp){
   logger.error(exp);
   throw exp;
}
```

12.4 USING FINALLY STATEMENT

Java finally block is a block that is used *to* execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block

Syntax

```
try
{
………
}
catch (……)
{
……….
}
finally
```

{

………

}

Example:

```
class TestFinallyBlock{
public static void main(String args[]){
 try{
  int data=25/5;
  System.out.println(data);
 }
 catch(NullPointerException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");
 }  }
```

UNIT XIII: INTRODUCTION

13.1 PREPARING TO WRITE APPLETS

Until now we have been creating simple Java application program with a single main() method that created objects, set instance variables and ran methods. To write any applet, we will need to know:

1. When to use applets.
2. How an applet works,
3. What sort of features an applet has, and
4. Where to start, when we first create our own applet.

The following are the steps that are involved in developing and testing and applet.

1. Buliding an applet code(.java file)
2. Creating an executable applet(.class file)
3. Designing a web page using HTML
4. Preparing <Applet Tag>
5. Incorporating <Applet> tag into the web page.
6. Creating HTML file.
7. Testing the applet code.

To building the applet code two classes of java library are essential namely Applet and Graphics. The Applet class is contained in java.applet package provides life and beehaviour to the applet through its methods such as int(), start() and paint(). Unlike with applications, where java calls the main() method directly to initiate the execution of the program, when an applet is

loaded java automatically calls a series of Applet class methods for starting running and stopping the applet code. The Applet class therefore maintains the life cycle of an applet.

public void paint(Graphics g)

Example:

```
import java.awt.*;

import java.applet.*;

……………….

………………

public class applet classname extends Applet

{

…………………

  ................................statements

  public void paint(Graphics g)

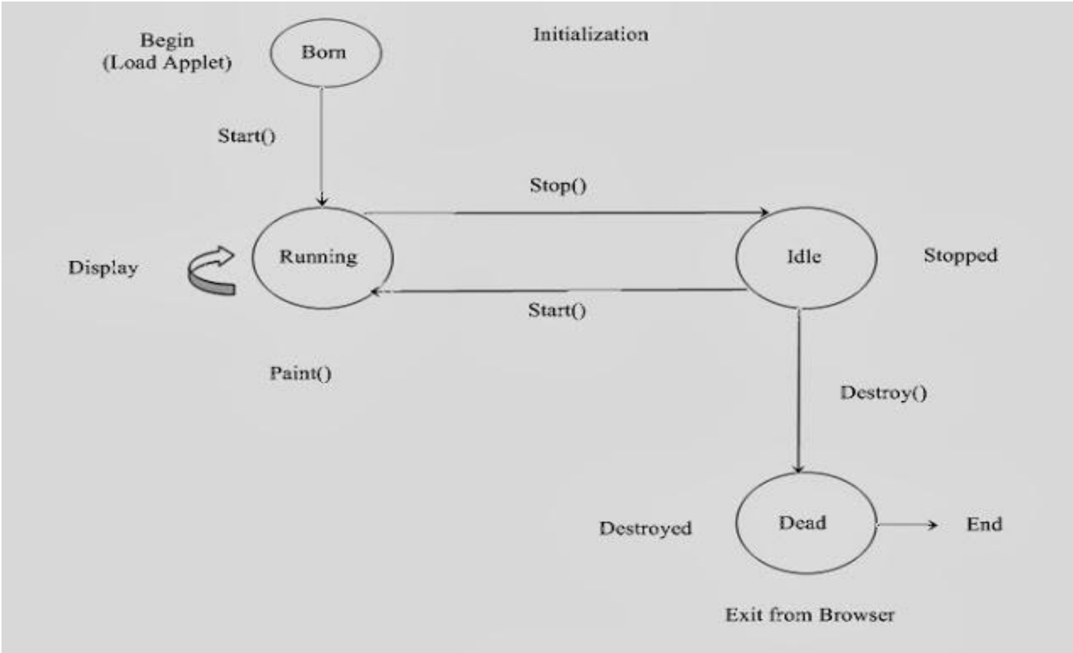
{

    ..........................//Applet operations code

}
  ......................
  .....................
}
```

13.2 APPLET LIFE CYCLE

An applet is a window based event driven program and it waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by applet. Applet should not enter a "mode" of operation in which it maintains control for an extended period. In these situations you must start an additional thread of execution.



Applet Life Cycle

Syntax

```
public class AppProgram extends Applet

{

 public void init()

{
   //initializtion
 }

 public void start()

{

   //start or resume execution

}

 public void stop()

{

   //suspend execution
```

```
}
  public void destroy()
{
    //perform shutdown activities
}
  public void paint(Graphics g)
{
    //redisplay contents of window
}
}
```

Initialization State

When the browser downloads an HTML page containing applets, it creates an instance for each of the Applet classes, using the no arg constructor. Applet must have a no argument constructor otherwise it cannot be loaded. Initialization can be done through init(). The init() method is the first method to be called. It is used to initialize the applet each time it is reloaded. Applets can be used for setting up an initial state, loading images or fonts, or setting parameters.

Syntax:

```
public void init()
{
  //code here
}
```

Running State

Immediately after calling init(), the browser calls the start() method. start() is also called when user returns to an HTMLpage that contains the applet. So, it ht user leaves a web page and come back, the applet resumes execution at start(). So, when the applet called the start() Method it is called its running state. Staring can also occur if the applet is already in "stopped" (idle) state.

Syntax:

```
public void start()
{
………..
(Action) ______________________
}
```

Idle or Stopped State

When we leave the page containing the currently running applet, then it stop running and becomes idle. We can also do so by calling stop() Method explicitly.

Syntax:

```
public void stop()
{
………..
(Action) ______________________
}
```

Dead State

When an applet is completely removed from the Memory, it is called dead. This occurs automatically by invoking the destroy() method when we quit the browser Like initialization, dead state occurs only once in the applet's life cycle

Syntax:

```
public void destory()
{
………..
(Action) ______________________
}
```

Display State

Painting is how an applet displays something on screen-be it text, a line, a colored background, or an image. The paint() method is used for displaying anything on the applet paint() method takes an argument, an instance of class graphics. The code given can be as follows:

Syntax:

```
public void paint(Graphics g)
{
}
```

13.3 APPLET TAG

The <Applet...< tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. The ellipsis in the tag <Applet...> indicates that it contains certain attributes that must specified. The <Applet> tag given below specifies the minimum requirements to place the Hellojava applet on a web page.

```
<Applet
Code=Hellojava.class
width=400
Height=200>
</Applet>
```

The applet tag discussed above specified the three things:
1) Name of the applet
2) Width of the applet (in pixels)
3) Height of the applet (in pixels)

This HTML code tells the browser to load the compiled java applet Hellojava.class, which is in the same directory as this HTML file. It also specifies the display area for the applet output as 400 pixels width and 200 pixels height. We can make this display area appear in the center of screen by using the CENTER tags as showsn below:

```
<CENTER>
<Applet>
-------------
-------------
-------------
</Applet>
</CENTER>
```

13.4 ADDING APPLET TO A HTML FILE

To execute an applet in a web browser, you need to write a short HTML text file that contains the appropriate APPLET tag. Here is the HTML file that executes **SimpleApplet**:

```
<Applet code = Hellojava.class width = 400 Height = 200 >
</Applet>
```

The width and height attributes specify the dimensions of the display area used by the applet. After you create this file, you can execute the HTML file called RunApp.html (say) on the command line.

```
c:\>appletviewer RunApp.html
```

13.5 RUNNING THE APPLET

To execute an applet with an applet viewer, you may also execute the HTML file in which it is enclosed, eg.

```
c:\>appletviewer RunApp.html
```

Execute the applet the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the applet tage within the comment and execute your applet.

Example :

The Following will be saved with named FirstJavaApplet.java:

```
import java.awt.*;

import java.applet.*;

public class FirstJavaApplet extends Applet

{

    public void paint(Graphics g)

      {

         g.drawString("My First Java Applet Program",100,100);

      }
```

```
}
```

After creating FirstJavaApplet.java file now you need create FirstJavaApplet.html file as shown below:

```
<HTML>

<HEAD>

 <TITLE>My First Java Applet Program</TITLE>

</HEAD>

<BODY>

 <APPLET Code="FirstJavaApplet.class" Width=300 Height=200>

 </APPLET>

</BODY>

</HTML>
```

OUTPUT:



UNIT XIV: THE GRAPHICS CLASS

14.1 LINES AND RECTANGLES

LINES:

In order to draw a line, you need to use the drawLine method of the Graphics class. This method takes four parameters, the starting *x* and *y* coordinates and the ending *x* and *y* coordinates.

Let's create a paint method that we will be adding to. We've already created the main method that runs the code, so we can simply add our shapes as we go.

Example:

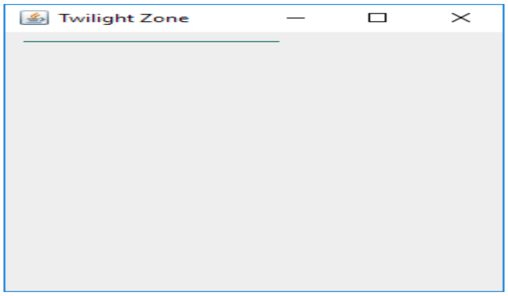
```
public void paint(Graphics g)
{ //custom color
  String hexColor = new String("0x45e5B");
  g.setColor(Color.decode(hexColor));
```

```
//draw a line (starting x,y; ending x,y)

g.drawLine(10, 10, 40, 10);

}
```

Output



RECTANGLES:

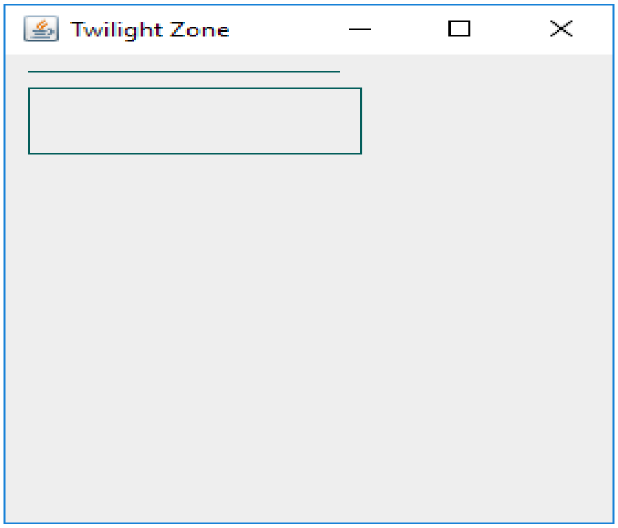
For our line, we used the Graphics class. We can make use of the newer **Graphics2D** class, which allows some more options when it comes to 2-D shapes, including thickness, anti-aliasing, etc. Add code to the beginning of the paint method to create a Graphics 2D instance that casts the Graphics class to Graphics2D:

```
//draw rectangle

g2.drawRect(10, 20, 150, 40);

g2.setColor(Color.decode(hexColor));
```

Output



14.2 CIRCLES AND ELLIPSES

The Graphics class does not have any method for circles or ellipses. However, the drawOvel() method can be used to draw a circle or an ellipse.

Like rectangle methods, the drawOval() method draws outline of an oval, and the fillOval() method draws a solid oval.

Example:

```
import java.awt.*;

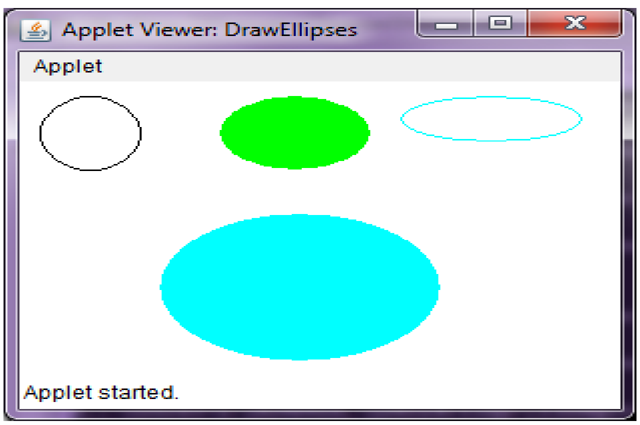
import java.applet.*;

 public class DrawEllipses extends Applet

{
```

```
public void paint(Graphics g)
{
 g.drawOval(10, 10, 50, 50);
 g.setColor(Color.GREEN);
 g.fillOval(100, 10, 75, 50);
 g.setColor(Color.cyan);
 g.drawOval(190, 10, 90,30);
 g.fillOval(70, 90, 140, 100);
}
}
```

OUTPUT



14.3 DRAWING ARCS

A segment of an oval is an arc. The arc angle can be positive (sweeps anti-clockwise) or negative (sweeps clockwise). As with other figures, the arcs can be outline or solid.
Supporting methods from java.awt.Graphics class

1. void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle): draws an outline arc.
2. void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle): draws a filled (solid) arc.

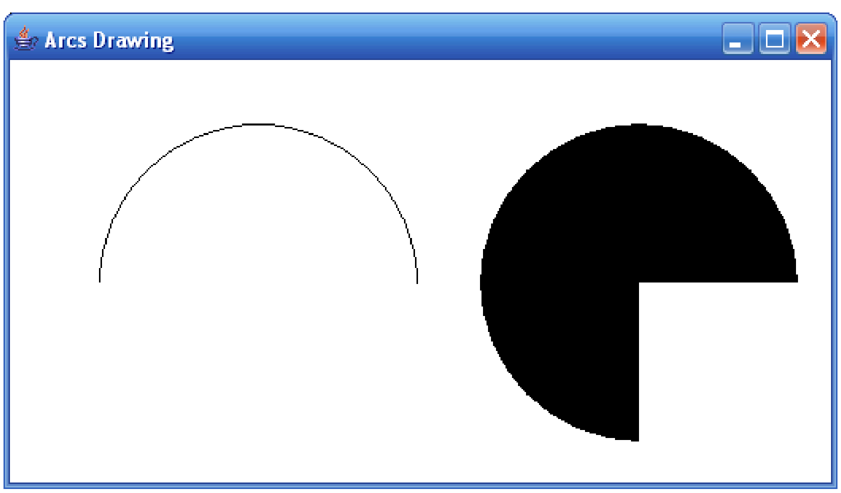
where

- x and y: Indicates the x and y coordinates where the arc is to be drawn
- width and height: Indicates the width and height of the arc
- startAngle: Dictates the starting angle of the arc
- arcAngle: The angular extent of the arc (relative to the start angle)

Example:

```
import java.awt.*;
public class ArcsDrawing extends Frame
{
public ArcsDrawing()
{
 setTitle("Arcs Drawing");
    setSize(525, 300);
    setVisible(true);
  }
  public void paint(Graphics g)
  {
    g.drawArc(60, 70, 200, 200, 0, 180);
    g.fillArc(300, 70, 200, 200, 0, 270);
  }
  public static void main(String args[])
  {
    new ArcsDrawing();}}
```

Output:



14.4 DRAWING POLYGONS

Polygon is a closed figure with finite set of line segments joining one vertex to the other. The polygon comprises of set of (x, y) coordinate pairs where each pair is the vertex of the polygon. The side of the polygon is the line drawn between two successive coordinate pairs and a line segment is drawn from the first pair to the last pair.

We can draw Polygon in java applet by three ways :

drawPolygon(int[] x, int[] y, int numberofpoints) : draws a polygon with the given set of x and y points.

Example:

```
import java.awt.*;

import javax.swing.*;

public class poly extends JApplet

{   // called when applet is started

public void init()

{  // set the size of applet to 300, 300

setSize(200, 200);

show();

} // invoked when applet is started

public void start()

{

}  // invoked when applet is closed

public void stop()

{

}

public void paint(Graphics g)

{   // x coordinates of vertices

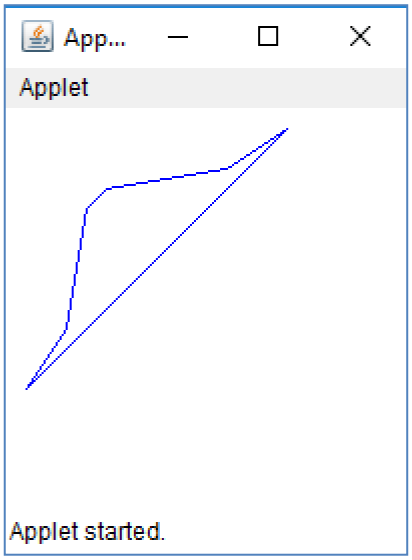
int x[] = { 10, 30, 40, 50, 110, 140 };   // y coordinates of vertices

int y[] = { 140, 110, 50, 40, 30, 10 };  // number of vertices

int numberofpoints = 6;                  // set the color of line drawn to blue
```

```
g.setColor(Color.blue);                    // draw the polygon using drawPolygon function
g.drawPolygon(x, y, numberofpoints);
}
}
```

Output:



14.5 LINE GRAPHS

We can design applets to draw line graphs to illustrate graphically the relationship between two variables

```
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;

public class LineChart_AWT extends ApplicationFrame {
```

```
public LineChart_AWT( String applicationTitle , String chartTitle ) {
  super(applicationTitle);
  JFreeChart lineChart = ChartFactory.createLineChart(
    chartTitle,
    "Years","Number of Schools",
    createDataset(),
    PlotOrientation.VERTICAL,
    true,true,false);

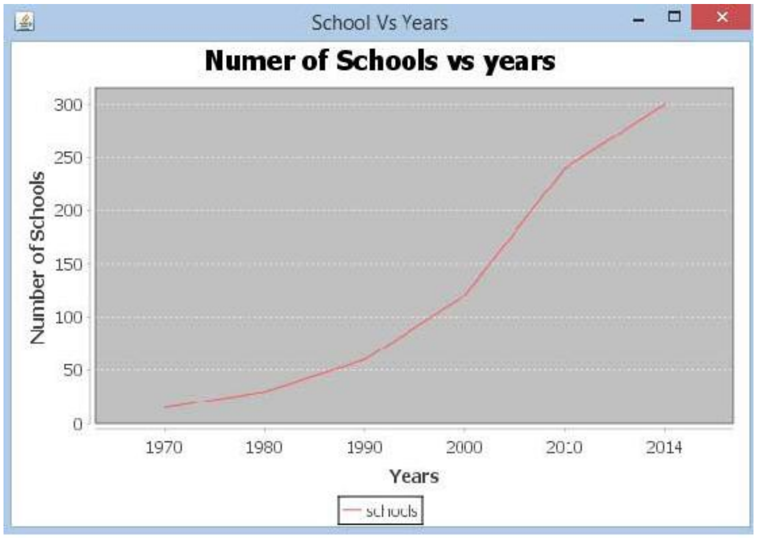
  ChartPanel chartPanel = new ChartPanel( lineChart );
  chartPanel.setPreferredSize( new java.awt.Dimension( 560 , 367 ) );
  setContentPane( chartPanel );
}

private DefaultCategoryDataset createDataset( ) {
  DefaultCategoryDataset dataset = new DefaultCategoryDataset( );
  dataset.addValue( 15 , "schools" , "1970" );
  dataset.addValue( 30 , "schools" , "1980" );
  dataset.addValue( 60 , "schools" ,  "1990" );
  dataset.addValue( 120 , "schools" , "2000" );
  dataset.addValue( 240 , "schools" , "2010" );
  dataset.addValue( 300 , "schools" , "2014" );
  return dataset;
}

public static void main( String[ ] args ) {
  LineChart_AWT chart = new LineChart_AWT(
    "School Vs Years" ,
    "Numer of Schools vs years");

  chart.pack( );
  RefineryUtilities.centerFrameOnScreen( chart );
  chart.setVisible( true );
}}
```

Output



Model Question Paper

B.Sc., Degree Examinations

Course: **JAVA PROGRAMMING**

Duration: 3 hours

Max. Marks: 100

PART – A

Answer all the questions **10 x 2 = 20 Marks**

1. Define Classes and Objects.
2. What is the output for the following program?

```
public class Demo
{   public static void main(String[] args)
    {
        int x=89+15>15?15:30;
        System.out.println(x);
    }
}
```

3. What is ‘final’ class ?
4. What is meant by “Instance Variable”?
5. Define Packages in Java.
6. How to compare two Strings in Java?
7. Define multithreading?
8. Define abstract class?
9. What is the role of init () method under applets?
10. State the syntax of drawline method.

PART – B

Answer all the questions **5 x 5 = 25 Marks**

11. (a) Elucidate “Command-line arguments” with a suitable example.

(OR)

(b) Write a Java program to find biggest among two numbers without using relational operators.

12. (a) Write a program to find maximum and minimum element in an array.

(OR)

(b) Explain Dynamic Method Dispatch with suitable example.

13. (a) Explain Multilevel inheritance with program example.

(OR)

(b) Discuss the Life cycle of a Thread.

14. (a) Illustrate the usage of ‘super’ keyword

(OR)

(b) Write a java program for exception handling.

15. (a) Discuss ‘Applet life cycle’.

(OR)

(b) Write an applet program to illustrate the Choice control.

PART – C

Answer all three questions **3 x 10 = 30 Marks**

16. Explain various operators in java with suitable examples. (10)

17. Write a program using String Class to sort a list of Strings. (10)

18. Explain in detail about Synchronization. **(10)**

19. (a) Write a program for student marklist preparation using single inheritance. (5)
    (b) Explain the usage of function overloading with example. 5)

20. Explain the attributes of Applet HTML tag and write the syntax for any two applet graphical methods with examples.(10)

B.Sc., Degree Examinations

Course: **JAVA PROGRAMMING**

PART – A

Answer all the questions **10 x 2 = 20 Marks**

1. Define the term ‘separators’ in Java.
2. Why the main () method is declared as static?
3. Define the term “Entry controlled loop”.
4. Illustrate the purpose of final keyword.
5. Define Packages.
6. List out any two methods of String Class.
7. State the need of ‘finally’ block.
8. What is the purpose of HTML in Applet?
9. List out any two methods of Graphics Class.
10. What is the purpose of FileNameFilter Class?

PART – B

Answer all the questions **5 x 5 = 25 Marks**

11. a) Write a program to find the factorial of a given number.

    Or

    b) Elucidate “Command-line arguments” with a suitable example.

12. a) Explain Dynamic Method dispatch with program.

    Or

    b) Discuss the importance of ‘this’ keyword with example.

13. a) Explain Multilevel inheritance with program example.

    Or

    b) List the operators supported by java.

14. a) Explain Applet Tags with example.

    Or

    b) Write a program using String Class to sort a list of Strings.

15. a) Discuss the Applet life cycle with neat sketch.

    Or

    b) Write a java program for exception handling.

PART – C

Answer any five questions **3 x 10 = 30 Marks**

16. Discuss Java Program Structure. (10)

17. Discuss the term “Overloading Methods” with a program example. (10)

18. Create a package ‘arith’ with an interface ‘Arithmetic’ and declare the methods for performing the operations addition, difference, product and division. Import this package into a java file ‘ArithDemo’ and implement all the above declared methods. (10)

19. Explain Hierarchical inheritance with a program example. . (10)

20. Brief the use of ‘Super’ keyword with program examples. (10)

SELF ASSESSMENT EXERCISES

BASIC JAVA PROGRAM

EX.1 TO CREATE A BASIC JAVA PROGRAMS

SOLUTION

/* This is a simple Java program.

```
FileName : "HelloWorld.java". */
class HelloWorld
{
	// Your program begins with a call to main().
	// Prints "Hello, World" to the terminal window.
	public static void main(String args[])
	{
		System.out.println("Hello, World");
	}
}
```

OUTPUT:

Hello, World

JAVA VIRTUAL MACHINE

EX.2 TO CREATE A JAVA PROGRAM OBJECT CREATED BY JVM TO REPRESENT

SOLUTION

```
// A Java program to demonstrate working of a Class type
// object created by JVM to represent .class file in
```

```
// memory.
import java.lang.reflect.Field;
import java.lang.reflect.Method;
// Java code to demonstrate use of Class object created by JVM
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student();
        // Getting hold of Class object created  by JVM.
        Class c1 = s1.getClass();

    // Printing type of object using c1.
        System.out.println(c1.getName());
        // getting all methods in an array
        Method m[] = c1.getDeclaredMethods();
        for (Method method : m)
            System.out.println(method.getName());

        // getting all fields in an array
        Field f[] = c1.getDeclaredFields();
        for (Field field : f)
            System.out.println(field.getName());
    }
}
// A sample class whose information is fetched above using
// its Class object.
class Student
```

```
{
        private String name;
        private int roll_No;
        public String getName() { return name; }
        public void setName(String name) { this.name = name; }
        public int getRoll_no() { return roll_No; }


public void setRoll_no(int roll_no) {
                this.roll_No = roll_no;
        }
}
```

OUTPUT:

```
Student
getName
setName
getRoll_no
setRoll_no
name
roll_No
```

DATA TYPES

EX 3: TO CREATE A JAVA PROGRAM TO DEMONSTRATE PRIMITIVE DATA TYPES IN JAVA

SOLUTION:

```
class GeeksforGeeks
{
```

```
public static void main(String args[])
{
        // declaring character
        char a = 'G';

        // Integer data type is generally
        // used for numeric values
        int i=89;
        // use byte and short if memory is a constraint
        byte b = 4;

        // this will give error as number is
        // larger than byte range
        // byte b1 = 7888888955;

        short s = 56;

        // this will give error as number is
        // larger than short range
        // short s1 = 87878787878;


        // by default fraction value is double in java
        double d = 4.355453532;

        // for float use 'f' as suffix
        float f = 4.7333434f;
```

```
        System.out.println("char: " + a);
        System.out.println("integer: " + i);
        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
    }
}
```

OUTPUT:

```
char: G
integer: 89
byte: 4
short: 56
float: 4.7333436
double: 4.355453532
```

TYPES CONVERSION

EX 4.1: TO CREATE A JAVA PROGRAM AUTOMATIC TYPES CONVERSION

SOLUTION:

```
class Test
```

```
{
	public static void main(String[] args)
	{
		int i = 100;
		//automatic type conversion
		long l = i;
		//automatic type conversion
		float f = l;
		System.out.println("Int value "+i);
		System.out.println("Long value "+l);
		System.out.println("Float value "+f);
	}
}
```

OUTPUT:

```
Int value 100
Long value 100
Float value 100.0
```

EX 4.2: TO CREATE A JAVA PROGRAM EXPLICIT CONVERSION OF INT AND DOUBLE TO BYTE

SOLUTION:

```
class Test
{
```

```
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("Conversion of int to byte.");
        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }
}
```

OUTPUT:

```
Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b = 67
```

OPERATORS

EX 5.1: TO CREATE A JAVA PROGRAM USING ARITHMETIC OPERATORS

SOLUTION

```
public class operators {
```

```
public static void main(String[] args)
{
        int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
        String x = "Thank", y = "You";

        // + and - operator
        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));

        // + operator if used with strings
        // concatenates the given strings.
        System.out.println("x + y = " + x + y);

        // * and / operator
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));

        // modulo operator gives remainder
        // on dividing first operand with second
        System.out.println("a % b = " + (a % b));

        // if denominator is 0 in division
        // then Arithmetic exception is thrown.
        // uncommenting below line would throw
        // an exception
        // System.out.println(a/c);
```

```
	}
}
```

OUTPUT:

```
a + b = 30
a - b = 10
x + y = ThankYou
a * b = 200
a / b = 2
a % b = 0
```

EX 5.2: TO CREATE A JAVA PROGRAM USING RELATIONAL OPERATORS

SOLUTION

```
public class operators {
	public static void main(String[] args)
```

```
{
    int a = 20, b = 10;
    String x = "Thank", y = "Thank";
    int ar[] = { 1, 2, 3 };
    int br[] = { 1, 2, 3 };
    boolean condition = true;

    // various conditional operators
    System.out.println("a == b :" + (a == b));
    System.out.println("a < b :" + (a < b));
    System.out.println("a <= b :" + (a <= b));
    System.out.println("a > b :" + (a > b));
    System.out.println("a >= b :" + (a >= b));
    System.out.println("a != b :" + (a != b));
    // Arrays cannot be compared with  relational operators because objects
    // store references not the value
    System.out.println("x == y : " + (ar == br));
    System.out.println("condition==true :"+ (condition == true));
}  }
```

OUTPUT:

```
a == b :false
a < b :false
a <= b :false
a > b :true
a >= b :true
a != b :true
```

```
x == y : false
condition==true :true
```

EXPRESSION

EX 6: TO CREATE A JAVA PROGRAM REPLACING CONDITIONAL OPERATOR WITH IF ELSE OR VICE VERSA

SOLUTION

```
// A Java program to demonstrate that we should be careful
```

```
// when replacing conditional operator with if else or vice
// versa
import java.io.*;
class GFG
{
	public static void main (String[] args)
	{
		// Expression 1 (using ?: )
		// Automatic promotion in conditional expression
		Object o1 = true ? new Integer(4) : new Float(2.0);
		System.out.println(o1);
		// Expression 2 (Using if-else)  No promotion in if else statement
		Object o2;
		if (true)
			o2 = new Integer(4);
		else
			o2 = new Float(2.0);
		System.out.println(o2);
	}
}
```

Output:

```
4.0
4
```

BRANCH LOOPING

EX 7.1: TO CREATE A JAVA PROGRAM TO ILLUSTRATE IF-ELSE-IF LADDER

SOLUTION

```
class ifelseifDemo
```

```
{
    public static void main(String args[])
    {
        int i = 20;
        if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
            System.out.println("i is 15");
        else if (i == 20)
            System.out.println("i is 20");
        else
            System.out.println("i is not present");
    }
}
```

Output : i is 20

EX 7.2: TO CREATE A JAVA PROGRAM TO ILLUSTRATE SWITCH-CASE

SOLUTION:

```
// Java program to illustrate switch-case
class SwitchCaseDemo
{
```

```
    public static void main(String args[])
    {
        int i = 9;
        switch (i)
        {
        case 0:
            System.out.println("i is zero.");
            break;
        case 1:
            System.out.println("i is one.");
            break;
        case 2:
            System.out.println("i is two.");
            break;
        default:
            System.out.println("i is greater than 2.");
        }
    }
}
```

OUTPUT:

i is greater than 2.

CLASS AND OBJECTS

EX 8: TO CREATE A JAVA PROGRAM USING CLASS AND OBJECTS

SOLUTION

```
// Class Declaration
```

```
public class Dog
{
   // Instance Variables
   String name;
   String breed;
   int age;
   String color;

   // Constructor Declaration of Class
   public Dog(String name, String breed,
               int age, String color)
   {
      this.name = name;
      this.breed = breed;
      this.age = age;
      this.color = color;
   }

   // method 1
   public String getName()
   {
      return name;
   }

   // method 2
   public String getBreed()
   {
      return breed;
   }

   // method 3
   public int getAge()
   {
      return age;
   }

   // method 4
   public String getColor()
   {
      return color;
   }

   @Override
   public String toString()
   {
      return("Hi my name is "+ this.getName()+
           ".\nMy breed,age and color are " +
           this.getBreed()+"," + this.getAge()+
```

```
            ","+ this.getColor());
    }

    public static void main(String[] args)
    {
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

Output:

```
Hi my name is tuffy.

My breed,age and color are papillon,5,white
```

INHERITANCE IN JAVA

EX 9: TO CREATE A JAVA PROGRAM ILLUSTRATE THE OF SINGLE INHERITANCE

SOLUTION

```
import java.util.*;
import java.lang.*;
```

```
import java.io.*;

class one
{
   public void print_geek()
   {
      System.out.println("Geeks");
   }
}
class two extends one
{
   public void print_for()
   {
      System.out.println("for");
   }
}              // Driver class
public class Main
{
   public static void main(String[] args)
   {
      two g = new two();
      g.print_geek();
      g.print_for();
      g.print_geek();
   }
}
```

OUTPUT

Greeks

For

Greeks

ARRAY

EX 10.1: TO CREATE A JAVA PROGRAM TO CREATING AN ARRAY OF INTEGERS

SOLUTION

// Java program to illustrate creating an array

```
// of integers,  puts some values in the array,
// and prints each value to standard output.

class GFG
{
    public static void main (String[] args)
    {
      // declares an Array of integers.
      int[] arr;

      // allocating memory for 5 integers.
      arr = new int[5];

      // initialize the first elements of the array
      arr[0] = 10;

      // initialize the second elements of the array
      arr[1] = 20;

      //so on...
      arr[2] = 30;
      arr[3] = 40;
      arr[4] = 50;

      // accessing the elements of the specified array
      for (int i = 0; i < arr.length; i++)
         System.out.println("Element at index " + i +
                                      " : "+ arr[i]);
    }
}
```

OUTPUT:

```
Element at index 0 : 10
Element at index 1 : 20
Element at index 2 : 30
Element at index 3 : 40
Element at index 4 : 50
```

EX 10.2: TO CREATE A JAVA PROGRAM TO CREATING AN MULTIDIMENSIONAL ARRAYS

SOLUTION

```
class multiDimensional
{
   public static void main(String args[])
   {
```

```
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++)
        {
            for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

            System.out.println();
        }
    }
}
```

OUTPUT:

```
2 7 9

3 6 1

7 4 2
```

INTERFACES

EX 11: TO CREATE A JAVA PROGRAM TO DEMONSTRATE WORKING OF INTERFACE

SOLUTION

```
// Java program to demonstrate working of
// interface.
import java.io.*;
// A simple interface
interface in1
{
	// public, static and final
	final int a = 10;
	// public and abstract
	void display();
}
// A class that implements interface.
class testClass implements in1
{
	// Implementing the capabilities of  interface.
	public void display()
	{
		System.out.println("Geek");
	}

	// Driver Code
	public static void main (String[] args)
	{
		testClass t = new testClass();
		t.display();
		System.out.println(a);
```

```
    }
}
```

OUTPUT:

```
Geek
10
```

PACKAGES AND MULTITHREADING PROGRAMMING

EX 12: TO CREATE A JAVA PROGRAM TO PACKAGES AND MULTITHREADING PROGRAMMING

SOLUTION

```
public class Test implements Runnable
```

```
{
   public void run()
   {
      System.out.printf("GFG ");
      System.out.printf("Geeks ");
   }
   public static void main(String[] args)
   {
      Test obj = new Test();
      Thread thread = new Thread(obj);

      thread.start();
      System.out.printf("Geeks ");
      try
      {
         thread.join();
      }
      catch (InterruptedException e)
      {
         e.printStackTrace();
      }
      System.out.println("for ");
   }
}
```

OUTPUT

a) GFG Geeks Geeks for
b) Geeks GFG Geeks for
c) Either a or b
d) Both a and b together

Ans. (c)
Explanation: From the statement “thread.start()”, we have two threads Main thread and “thread” thread. So either “GFG” can be printed or “Geeks”, depend on which thread, thread scheduler schedule.
For (a), the parent thread after calling start() method is paused and the thread scheduler schedules the child thread which then completes its execution. Following this, the parent thread is

scheduled. For (b), the parent thread calls start() method but continues its execution and prints on the console. When join() method is called, the parent thread has to wait for its child to complete its execution. Thread scheduler schedules child thread while the parent waits for the child to complete.

MULTI CATCH

EX 13: TO CREATE A JAVA PROGRAM MULTIPLE CATCH BLOCKS FOR MULTIPLE EXCEPTIONS

SOLUTION

```
import java.util.Scanner;
public class Test
{
   public static void main(String args[])
```

```
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
        catch (ArithmeticException ex)
        {
            System.out.println("Arithmetic " + ex);
        }
        catch (NumberFormatException ex)
        {
            System.out.println("Number Format Exception " + ex);
        }
    }
}
```

Input 1:

```
GeeksforGeeks
```

Output 2:

```
Exception encountered java.lang.NumberFormatException:

For input string: "GeeksforGeeks"
```

Input 2:

```
0
```

Output 2:

```
Arithmetic Exception encountered java.lang.ArithmeticException: / by zero
```

EXTENDING THE THREAD CLASS

EX 14.1: TO CREATE A JAVA PROGRAM THREAD CREATION BY EXTENDING THE THREAD CLASS
SOLUTION

```
/ Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread
```

```
{
   public void run()
   {
      try
      {
         // Displaying the thread that is running
         System.out.println ("Thread " +
              Thread.currentThread().getId() +
              " is running");

      }
      catch (Exception e)
      {
         // Throwing an exception
         System.out.println ("Exception is caught");
      }
   }
}

// Main Class
public class Multithread
{
   public static void main(String[] args)
   {
      int n = 8; // Number of threads
      for (int i=0; i<8; i++)
      {
         MultithreadingDemo object = new MultithreadingDemo();
         object.start();
      }
   }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
```

```
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

EX 14.2: TO CREATE A JAVA PROGRAM THREAD CREATION BY IMPLEMENTING THE RUNNABLE INTERFACE

SOLUTION

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable
{
    public void run()
```

```
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

OUTPUT :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
```

```
Thread 14 is running
Thread 15 is running
```

SYNCHRONIZED IN JAVA

EX 15: TO CREATE A JAVA PROGRAM TO DEMONSTRATE WORKING OF SYNCHRONIZED

SOLUTION

// A Java program to demonstrate working of

```
// synchronized.
import java.io.*;
import java.util.*;

// A Class used to send a message
class Sender
{
   public void send(String msg)
   {
      System.out.println("Sending\t"  + msg );
      try
      {
         Thread.sleep(1000);
      }
      catch (Exception e)
      {
         System.out.println("Thread  interrupted.");
      }
      System.out.println("\n" + msg + "Sent");
   }
}

// Class for send a message using Threads
class ThreadedSend extends Thread
{
   private String msg;
   private Thread t;
   Sender  sender;

   // Recieves a message object and a string
   // message to be sent
   ThreadedSend(String m,  Sender obj)
   {
      msg = m;
      sender = obj;
   }

   public void run()
   {
      // Only one thread can send a message
      // at a time.
      synchronized(sender)
      {
         // synchronizing the snd object
         sender.send(msg);
      }
   }
}
```

```
// Driver class
class SyncDemo
{
    public static void main(String args[])
    {
        Sender snd = new Sender();
        ThreadedSend S1 =
            new ThreadedSend( " Hi " , snd );
        ThreadedSend S2 =
            new ThreadedSend( " Bye " , snd );

        // Start two threads of ThreadedSend type
        S1.start();
        S2.start();

        // wait for threads to end
        try
        {
            S1.join();
            S2.join();
        }
        catch(Exception e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

OUTPUT:

```
Sending    Hi


 Hi Sent
Sending    Bye


 Bye Sent
```

JAVA APPLET

EX 16: TO CREATE A JAVA PROGRAM TO CREATE A JAVA APPLET BASIC PROGRAM

SOLUTION

```
// A Hello World Applet
// Save file as HelloWorld.java
```

```
import java.applet.Applet;
import java.awt.Graphics;

/*
<applet code="HelloWorld" width=200 height=60>
</applet> xa
*/

// HelloWorld class extends Applet
public class HelloWorld extends Applet
{
    // Overriding paint() method
    @Override
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 20, 20);
    }

}
```

With this approach, first compile HelloWorld.java file and then simply run below command to run applet :

```
appletviewer HelloWorld
```

OUTPUT:

```
appletviewer RunHelloWorld.html
```



DRAWLINE

EX 17: TO CREATE A JAVA PROGRAM TO DRAW A RECTANGLE USING DRAWLINE

SOLUTION

// Java Program Draw a rectangle

```
// using drawLine(int x, int y, int x1, int y1)
import java.awt.*;
import javax.swing.*;

public class rectangle extends JApplet {

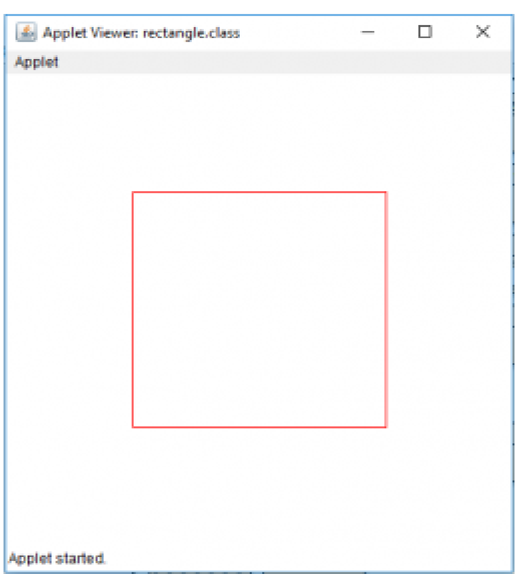
    public void init()
    {
        // set size
        setSize(400, 400);

        repaint();
    }

    // paint the applet
    public void paint(Graphics g)
    {
        // set Color for rectangle
        g.setColor(Color.red);

        // draw a rectangle by drawing four lines
        g.drawLine(100, 100, 100, 300);
        g.drawLine(100, 300, 300, 300);
        g.drawLine(300, 300, 300, 100);
        g.drawLine(300, 100, 100, 100);
    }
}
```

Output :



DRAWOVAL

EX 18: TO CREATE A JAVA PROGRAM DRAW A ELLIPSE USING DRAWOVAL

SOLUTION

```
// java program to draw a ellipse
// using drawOval function.
import java.awt.*;
```

```
import javax.swing.*;

public class ellipse extends JApplet {

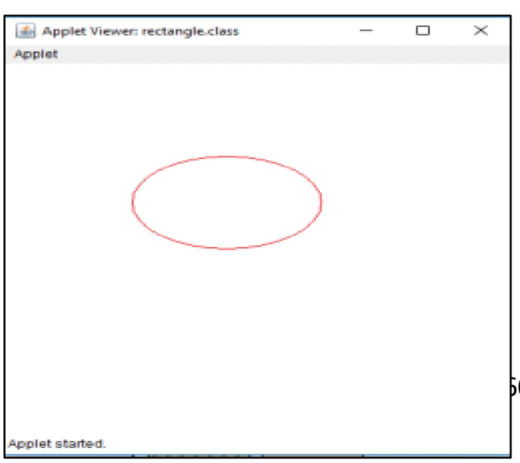
    public void init()
    {
        // set size
        setSize(400, 400);

        repaint();
    }

    // paint the applet
    public void paint(Graphics g)
    {
        // set Color for rectangle
        g.setColor(Color.red);

        // draw a ellipse
        g.drawOval(100, 100, 150, 100);
    }
}
```

OUTPUT **:**



DRAW POLYGON

EX 19: TO CREATE A JAVA PROGRAM DRAWS A POLYGON WITH THE GIVEN SET OF X AND Y POINTS.

SOLUTION

```
// Java program to draw polygon using
// drawPolygon(int[] x, int[] y, int numberofpoints)
// function
```

```
import java.awt.*;
import javax.swing.*;

public class poly extends JApplet {

   // called when applet is started
   public void init()
   {
      // set the size of applet to 300, 300
      setSize(200, 200);
      show();
   }

   // invoked when applet is started
   public void start()
   {
   }

   // invoked when applet is closed
   public void stop()
   {
   }

   public void paint(Graphics g)
   {
      // x coordinates of vertices
      int x[] = { 10, 30, 40, 50, 110, 140 };

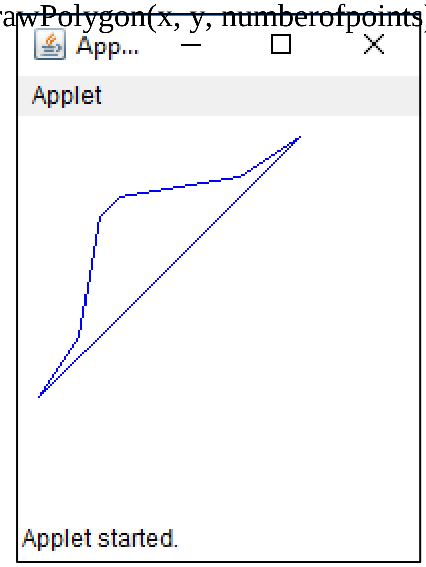
      // y coordinates of vertices
      int y[] = { 140, 110, 50, 40, 30, 10 };

      // number of vertices
      int numberofpoints = 6;

      // set the color of line drawn to blue
      g.setColor(Color.blue);

      // draw the polygon using drawPolygon function
      g.drawPolygon(x, y, numberofpoints);
   }
}
```

OUTPUT :



Further Readings

TEXT BOOK

1. Schildt,"The Complete Reference Java,TMH,11th edition,2018

2. Deitel & Deitel ,"JAVA How To Program ", Pearson Education Series,9th edition,2011.

3. Andrew Lee Rubinger & Bill Bruke, "Enterprise Java Beans ",O'Reilly Media, 6th Edn, 2010.
4. Phil Hanna, "JSP 2.0: The Complete Reference", 2nd edition, TMG, 2010.

5. Patrick Naughton, Herbert Schildt ,"JAVA2: The Complete Reference",TMH,5th edition , 2006.

6. George Reese, "Database Programming with JDBC & Java", 2nd Edition, O'Reilly Media,2000

7. Joseph O'Neil,"Teach Yourself JAVA", TMH,reprint 2001.

8. Holzner, S, JAVA 2 Swing, Servlets, JDBC & Java Beans Programming, Dream Tech Press, Fifth Edition, 2001.

9. Zukowski. J, Mastering JAVA2, Second Edition, BPB Publication, 1998.

10. Kenny Chu,"The Complete Reference Java",TMH,1997,7th edition